博士論文

ソフトウェアテストの自動化を目指した組み合わせテスト入力値限局手法とテストコード自動生成手法の考案

京都工芸繊維大学
工芸科学研究科 設計工学専攻

西浦 生成

2021

概要

ソフトウェアの品質を高め信頼性を保つことは重要な課題である．そのため，不具合の事前検出を目的としたソフトウェアテストが広く行われている．しかしソフトウェアテストに要求されるリソースは重く，自動化による作業の短縮化が望まれている．またテストプロセスには，テストケースの設計，テストの実行，不具合箇所の限局など多くの工程が含まれており，生産性の拡大のため全てのテスト工程を横断的に自動化できることが望まれる．しかし，組み合わせテスト設計ツールに代表されるようなテストケース設計の自動化や，継続的インテグレーションに代表されるテスト自動実行環境の整備など，部分における自動化は発展している一方で，未だ十分な自動化が実現されておらず，横断的なテスト自動化のボトルネックとなる部分も存在している．

こうしたソフトウェアテストプロセスの自動化に関連する課題から，本論文では入力値限局およびテストコード生成の 2 つのトピックに焦点を当て，問題の改善に取り組んだ．入力値限局とは，失敗したテストケースを含む組み合わせテストの結果から不具合を誘発する入力値の組み合わせを限局する操作を指す．限局結果からは開発者がソフトウェアシステムから不具合を取り除くための有益な情報が得られるため，自動修復技術の精度にも大きく関係する．またテストコードとは，テスト操作を自動的に行うために作成されたプログラムを指す．テストコードを使ったテストの自動実行は現在では主流となっている．

入力値限局における問題点として，テスト結果を分析し疑わしい組み合わせを推定する既存手法では，その精度に改善の余地がある．また限局の過程で不具合を誘発した可能性のある組み合わせを全て列挙する必要があるため，組み合わせ爆発により処理時間が膨大化することがある．さらに，既存手法は同一のテストケースの

結果が常に同一であることを前提とする一方，その前提に沿わない非決定的なテスト結果の存在が報告されており，既存手法では不可能な場合がある．

次にテストコードを利用した自動テスト実行における問題点として，テストコードの作成には相応の労力がかかる．そのためテストコードを自動生成できることが望まれている．しかし，既存のテストコード自動生成技術は単純かつ低品質な単体テストのみを生成対象としていることや，構造化された仕様書のようなデータの整備が必要であるといった限界がある．これに対し，OSS を対象とした先行調査によって，類似した機能を持つプロジェクトのテストコードを再利用することで目的のテストコードを用意できる可能性が示唆されている．しかし，実際にテストコードを自動的に再利用する方法は未だ開拓されていない．

本論文では，こうした入力値限局およびテストコード生成における課題を改善するための手法を提案する．またその評価や調査の結果を示すことで提案の有効性を明らかにする．

まず，ロジスティック回帰分析によって得られる回帰係数を疑惑値として利用でき，既存手法よりも高精度に入力値限局が可能であることを評価実験により明らかにした．また限局すべき組み合わせの回帰係数が高くなることに加え，その部分集合にも高い回帰係数が見られることを発見した．この特性を利用することで，限局すべき組み合わせの部分集合を先に推定し，それらの超集合として組み合わせを推定する手法を構築した．また実際のシステムに基づくテスト結果を用いた評価実験によって，処理を高速化しつつ，高い精度で実際の不具合誘発入力値を推定できることを示した．

次に，先行研究で報告されたテスト結果が非決定的になる要因のうち実行順序への依存性に焦点を絞り，非決定的なテスト結果に対応できるよう拡張された入力値限局手法を提案した．初めにテストの失敗を再現するテスト実行順序を特定し，それらを連続に実行して非決定性を排除することで，正しい限局結果を得ることが期待できる．実際のシステムに基づくテスト結果を用いた評価実験によって，提案手法の特定精度およびテストの追加実行回数の増加割合が好ましいものであることを示した．

最後に，OSS からの再利用による Java テストコードの自動生成手法の開発に取

り組んだ．まず，特定のテストメソッドを実行可能性を保ちつつ自動的に移植するために必要な条件を定義し，それに基づいてOSS中のテストメソッドが持つ依存関係等を調査することで，テストコードの再利用が現実的かつ有益であることを示した．加えて，既存テストコードの再利用によるテストコード自動生成モデルを実際に提案した．このモデルでは，テスト対象コードへの依存を分析し，機械的操作のみによって実行可能性を維持できるテストメソッドを検出することで，移植用のテストコードを自動的に生成する．さらに，自然言語処理技術によってテスト操作の意味的な適切さを担保する．また，このモデルをいくつかのAndroidアプリケーションに適用した事例によって，手法の有効性および課題を示した．

主要論文一覧

(1) 西浦生成，崔銀惠，水野修，”ロジスティック回帰分析を用いた組合せテストの不具合特定法の提案，”FOSE ソフトウェア工学の基礎研究会 (FOSE2016) 論文集，pp.243–244 (December 2016).

(2) Kinari Nishiura, Eun-Hye Choi, and Osamu Mizuno, ”Improving Faulty Interaction Localization Using Logistic Regression,” *Proc. of 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp.138–149 (July 2017).

(3) 西浦生成，崔銀惠，水野修，”機械学習による不具合組み合わせ特定への自動分類法の提案と評価，”ソフトウェアエンジニアリングシンポジウム 2017 論文集，pp.24–35 (August 2017).

(4) 西浦生成，崔銀惠，水野修，”機械学習による不具合組み合わせ特定への自動分類法の提案と評価，”情報処理学会論文誌, Vol.59, No.4，pp.1215–1224 (April 2018).

(5) 渡辺大輝，西浦生成，水野修，”不具合誘発パラメータ組み合わせ特定三手法の比較評価，”ソフトウェア・シンポジウム 2018 論文集，pp.47–56 (June 2018).

(6) 近藤将成，崔恩瀞，西浦生成，水野修, ”リモートワークにおけるソフトウェア開発者間のコミュニケーション方法の調査，”ソフトウェアエンジニアリングシンポジウム 2020 論文集，(September 2020).

(7) Kai Yamamoto, Masanari Kondo, Kinari Nishiura, and Osamu Mizuno, ”Which Metrics Should Researchers Use to Collect Repositories: an Empir-

ical Study,” *Proc. of 2020 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp.458–466 (December 2020).

(8) 西浦生成，渡辺大輝，水野修，崔恩瀞，”組合せテストにおける実行順序に起因する非決定的不具合誘発要因特定法の提案，” 情報処理学会論文誌 Vol.62, No.4，pp.1008–1018 (April 2021).

(9) 西浦生成，水野修，崔恩瀞，”Java テストコードの再利用による自動生成に向けた移植可能なテストメソッドの調査，” 情報処理学会論文誌 Vol.62, No.4, pp.1019–1028 (April 2021).

(10) 頭川剛幸，近藤将成，西浦生成，水野修, ” 構成管理ツールにおける命令的操作が及ぼすソースコードレビューへの影響調査，” ソフトウェア・シンポジウム 2021 論文集，採録済 (June 2021).

謝辞

博士学位論文を提出するにあたって，多くの方々のご指導とご助力をいただきました.

本研究を進めるにあたり，ソフトウェア工学の研究分野や論文の作成に関する深い知識を授けていただくのみならず，研究に向かう姿勢や研究に関する困難克服のための具体的な方策まで，終始あたたかいご指導と激励を賜りました，指導教官である京都工芸繊維大学情報工学・人間科学系教授 水野 修 先生に心から感謝の意を表します.

また，論文審査委員会の副査を務めていただきました，同系教授 渋谷 雄 先生および 辻野 嘉宏 先生からは，この論文に対して貴重なコメントと有益な批評をいただきました．深くお礼を申し上げます.

さらに，私が後期博士課程で履修した多くの教科は，この論文の準備に大変役立ちました．同系教授 稲葉 宏幸 先生，梅原 大祐 先生，岡 夏樹 先生，平田 博章 先生，寶珍 輝尚 先生，桝田 秀夫 先生のご指導に感謝いたします.

加えて，当研究室の同系助教 崔 恩瀞 先生には，研究の進め方や枠組みについて明晰かつ丁寧なご指導をいただきました．崔 先生のご指導により，本研究にいっそう深く取り組むことができ，また本論文をより磨き上げることができたと感じております．謹んで感謝申し上げます.

また，これまで当研究室で共同研究を行ってきた 渡辺 大輝 氏，近藤 将成 氏，山本 凱 氏，頭川 剛幸 氏にも感謝いたします．彼らの協力は私の論文に欠かせないものでした．特に，渡辺 氏との共同研究は本論文において大きな役割を担いました．また当研究室のこれまでの他メンバーの皆様からも，常に刺激的な議論を頂き，精神的にも支えられました．ありがとうございます.

国立研究開発法人産業技術総合研究所 情報・人間工学領域 サイバーフィジカル

セキュリティ研究センターの 崔 銀惠 氏，森 彰 氏には，私の研究に対して親切なご協力と洞察に満ちたアドバイスをいただきました．また共同研究を通し，研究の楽しさと難しさを教えてくださいました．心より御礼申し上げます．

私の訪問留学を快く受け入れていただいた Queen's University の Ying Zou 教授にも格別の感謝を申し上げます．現地の研究グループの皆様とは私の研究アイデアについて貴重なご意見を交換でき，素晴らしい経験を積むことができました．

最後に，両親や多くの友人達から有益な助言や励ましを頂きました．心からお礼を申し上げます．

図一覧

表一覧

目次

第 1 章

序論

1.1 背景

ソフトウェアの品質を高め信頼性を保つことは重要な課題である．そのため，不具合の事前検出を目的としたソフトウェアテストが広く行われている．ソフトウェア開発において，ソフトウェアテストは製品の品質を維持するために重要な役割を果たす．しかし，ソフトウェアテストの実施には多くのリソースが必要となる．IPA/SEC が発行するソフトウェア開発白書 2018-2019 [69] によると，結合テストおよび総合テストに費やす平均工数割合の合計は新規開発で 36.9%，改良開発で 36.4%と大きな比重を占めている．またソフトウェアの複雑化及び高信頼化は進んできており，テスト作業はますます膨大化し，開発者の負担は大きくなっていくと考えられる．これらのことから，テスト作業の自動化による作業の最適化および短縮化は労力削減の面で重要な意味を持つ．

またテストプロセスには，テストケースの設計，テストの実行，不具合箇所の限

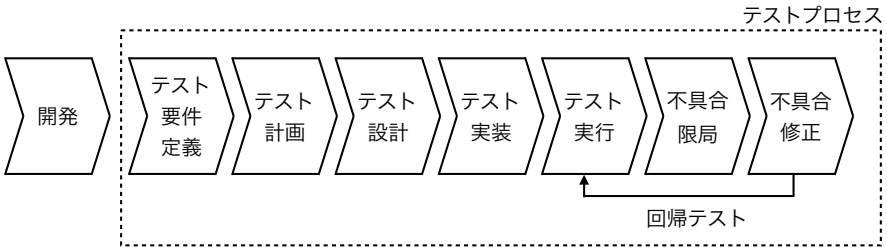

図 1.1: テストプロセスに含まれる作業工程

局など多くの工程が含まれており，生産性の拡大のため全てのテスト工程を横断的に自動化できることが望まれる．図 1.1 にテストプロセスに含まれる作業工程の流れを示す．しかし，組み合わせテスト設計ツールに代表されるようなテストケース設計の自動化や，継続的インテグレーションに代表されるテスト自動実行環境の整備など，部分における自動化は発展している一方で，未だ十分な自動化が実現されておらず，横断的なテスト自動化のボトルネックになっている部分も存在している．

1.2 研究目的

本論文では，テストプロセス自動化の障壁となる具体的な諸問題について取り組み，制限の緩和を目指す．本論文ではこのうち組み合わせテストの入力値限局およびテストコード生成の 2 つの研究分野に焦点を当て，問題の改善に取り組んだ．これら 2 つの研究分野では前述したテストケース設計の自動化やテスト自動実行環境の整備などに比べ，十分な品質を備えたデファクトスタンダードとなるような技術が確立されておらず，分野としても発展途上である．

1.2.1 組み合わせテストの入力値限局

組み合わせテストはテスト最適化の有力な手法として知られている [38, 49]．一般に，一つのテストケースはテスト対象のシステムの一つの動作のみをテストするため，さまざまな条件下でのテストを行うには複数のテストケースが必要となる．ただし，大規模なソフトウェアをテストする場合，複数の入力パラメータの相互作用

によって引き起こされる不具合を網羅的に検出するには膨大な数のテストケースが必要となり，そのすべてをテストすることは現実的ではない．こうした場合，組み合わせテストを使用することでテスト実行のコストを大幅に削減できる．組み合わせテストでは，特定の基準に対して入力パラメータがとる値のすべての組み合わせを網羅するようにテストケースを設計し，設計された一連のテストケースを使用してテストを実施する．組み合わせテストを用いることで，入力パラメータの相互作用に起因する不具合の発生を最小限のテストケースで検出することが可能になる．

しかし，組み合わせテスト技術を用いると，あるテストケースが失敗したときに，どの入力パラメータの組み合わせが不具合を引き起こしたのかを一意に特定することが難しい側面を持つ．これは，全ての入力値の組み合わせを完全に網羅したテストケース集合を用いた場合にはテストの失敗を再現する入力条件を一意に特定するための情報を欠落無く得ることができる一方，組み合わせテスト技術を用いた場合は一意な特定に至る情報が不十分となるためである．従って不具合を修正するためには，テストによる不具合の検出後，その不具合を引き起こした入力パラメータの組み合わせを特定する作業を別途行う必要がある．この作業は不具合誘発入力値組限局（Faulty Interaction Localization，あるいは Identifying Failure-Inducing Combinations）と呼ばれ，多くの先行研究が行われている [27,48,50,65,66]．これを本論文では単に入力値限局と呼称する．入力値限局の結果からは開発者がソフトウェアシステムから不具合を取り除くための有益な情報が得られるため，自動修復技術の精度にも大きく関係する．入力値限局を行う既存の一つの方法として，組み合わせテストを設計する段階で，一定の組み合わせ数に限定した入力値限局能力を持つテスト集合を設計する方法がある．しかし，シンプルな組み合わせテストと比べて数倍のテストケースが必要になるため，テストケース数最適化による恩恵が減少し，テストを運用する上でリスクが大きい．一方少例ながら，シンプルな組み合わせテストのテスト結果を分析し，不具合を誘発している可能性の高い組み合わせを推定することで効率的に入力値限局を行う方法も存在する．ただし，これらの入力値限局手法は推定的な手法であるがゆえに精度改善の余地がある．また，規模の大きいテストを扱う際に組み合わせ爆発を起こす場合があり，非現実的な処理時間となってしまうという問題を抱えている．さらに，既存手法は同一のテストケース

の結果が常に同一であることを前提とする一方で，その前提に沿わない非決定的なテスト結果の存在が報告されている．そのため既存の入力値限局手法では，こうした非決定的なテスト結果から正しい限局結果を導くことは非常に困難であり，対応が求められている．

1.2.2 テストコードの自動生成と再利用

テストコードとは，テスト操作を自動的に行うために作成されたプログラムを指す．かつてのソフトウェアテストでは手動による入力および出力の目視確認によってテストが行われていたが，この労力を軽減すべく，自動的にテストを行うためのプログラムであるテストコードが作成されるようになった [52]．テストコードを使った自動テスト実行は現在では主流となっている．テストコードには通常，テスト対象プログラムへの入力あるいは操作とその期待出力が記述され，実際の出力と期待出力を照合することで，動作が正しく行われているかを確認できる．テストコードを実行することで，テストを素早く繰り返し行うことができ，ソフトウェアテストの実行に伴うコストを軽減できる．

一方で，テストコードは基本的に手動で記述されるため，作成には相応の作業コストが要求される．この作業コストは自動テスト導入への障害となるほか，開発者の過労働やテストの作成不足による不具合発生を誘発する可能性がある．この問題に対し，テストコードの自動生成技術による緩和が期待されている．しかし，既存のテストコード自動生成技術 [24,40,61] は入力として何らかのドキュメントを必要とするか，でなければ非常に単純な単体テストのみしか作成できないという問題点を持つ．

これに対し，オープンソースソフトウェアを対象とした先行調査によって，類似した機能を持つプロジェクトのテストコードを再利用することで目的のテストコードを用意できる可能性が示唆されている．しかし，実際にテストコードを自動的に再利用する方法は未だ開拓されていない．既存のテストコードを再利用することができれば，過去に人手によって作られ使用されていた高品質なテストコードを手軽に生成できる可能性がある．そのため，既存のテストコードを自動的に再利用することで Java テストコードの自動生成を行う手法を開発することが求められている．

加えて，既存テストコードを再利用することで開発者の知識が不十分な場合にも，適切なテストシナリオやテストオラクルを伴うテストコードを生成できる可能性がある．テストシナリオとは，テスト対象をテストするための操作を指す．テストオラクルとは，特定のテストシナリオにおいてテスト対象に期待される動作あるいは出力を指す．テストオラクルはテスト対象から演繹的に導出することが困難であり，通常は人間の知識や経験によって導出される．そのため，テストオラクル問題はコンピュータマシンによる自律的なテストへの重大な障害として知られている．既存のテストコードを再利用することによって，他の開発者のアイデアに基づくテストシナリオおよびテストオラクルを自分のプロジェクトに適用でき，テストオラクル問題を解決できることが期待される．

1.3 主要な結果

本論文では，こうした入力値限局およびテストコード生成における課題を改善するためのいくつかの手法を提案する．またその評価や調査の結果を示すことで提案の有効性を明らかにする．

入力値限局手法に関する本研究の主要な結果を以下の 1，2，3 に示す．また，Java テストコードの再利用に関する本研究の主要な結果を以下の 4，5 に示す．

1. 入力値限局におけるロジスティック回帰を用いた分析精度の改善
2. 入力値限局における部分集合の推定による計算効率の改善
3. 入力値限局における実行順序に依存したテスト結果への拡張
4. OSS を対象とした Java テストコードの再利用可能性分析
5. 再利用に基づく Java テストコード自動生成手法の提案

1.3.1 入力値限局におけるロジスティック回帰を用いた分析精度の改善

不審度推定に独自のアルゴリズムを採用した既存研究 [4] に代わり，ロジスティック回帰分析によって不審度の推定を行う方法を開発した．これは機械学習手法として

知られるロジスティック回帰の学習メカニズムを利用した分析方法である．この分析方法を入力値限局に応用したのは本研究が初であり，この点で独創性がある．本手法では，テスト結果から得られる不具合を誘発した可能性のある全ての組み合わせを抽出してテストケースとの包含関係をエンコードし，ロジスティック回帰への入力とすることで各組み合わせの疑惑値として回帰係数が得られ，それを順位付ける．この提案モデルを FROGa と命名した．FROG は，"FIL based on Regression coefficient of lOGisic regression（ロジスティック回帰の回帰係数に基づいた入力値限局）"に由来する．第 4 章で提案する手法も同様にロジスティック回帰分析を使用するため，区別のためにこちらを FROGa，あちらを FROGb と呼称する．また実在するシステムを基にした生成した組み合わせテスト結果を使用し，提案手法が既存手法よりも高精度に入力値限局が可能であることを評価実験により明らかにした．実験では以下に示す 2 つの研究設問（Research Question，RQ）を設定し，それぞれの結果を得た．

RQ1 FROGa は BEN と比較して不具合誘発スキーマとしての疑わしさをランク付けする精度を向上させるか？

結果：FROGa は，BEN と比較して候補スキーマの疑わしさをランク付けする精度を大幅に向上させることができる．特に不具合誘発スキーマが 1 つのみ存在する場合，FROGa はほぼ確実にそのスキーマをランキングの頂点に位置付けることができる．また，入力値の多いシステムおよび網羅度の高い組み合わせテストを対象とした入力値限局では BEN の精度は低くなるが，FROGa の精度は高いまま保たれる．

RQ2 FROGa と BEN の時間コストに差はあるか？

結果：BEN と FROGa の時間コストの間には，明らかな違いはない．

1.3.2 入力値限局における部分集合の推定による計算効率の改善

FROGa に使用したロジスティック回帰分析の利用方法を応用することで，不具合を誘発する入力値組み合わせの部分集合となる入力値も同様に高い回帰係数を得られることが期待できる．この特性を利用することで，不具合を誘発する入力値組み合

わせの部分集合を先に推定し，それらの超集合として不具合を誘発する入力値組み合わせを推定する手法 FROGb を構築した．これによって，組み合わせ爆発を回避し，処理を効率化することが期待できる．こうした入力値限局における推定の効率化を目的とした研究は他に例がなく，目的，手法ともに強く新規性がある．FROGa が既存手法 BEN の順位付け手法のみを改善したものであるのに対して，FROGb は BEN および FROGa とは根本的に異なる方法で入力値限局を行うことに注意されたい．また実際のシステムに基づくテスト結果を用いた評価実験によって，処理を高速化しつつ，高い精度で実際の不具合誘発入力値を推定できることを示した．実験では以下に示す 2 つの研究設問を設定し，それぞれの結果を得た．

RQ3 FROGb は，BEN および FROGa に比べて非常に不審な候補スキーマを抽出するコストをどれだけ削減できるか？

結果：FROGb は，特に網羅次数の高い組み合わせテストを対象とした場合に，BEN や FROGa に比べて処理時間を大幅に短縮しながら，非常に疑わしい少数の候補スキーマのみを抽出することが可能である．

RQ4 FROGb は，非常に不審な候補スキーマとして不具合誘発スキーマをどれだけ正確に抽出できるか?

結果：FROGb は必ず全ての不具合誘発スキーマを非常に不審な候補スキーマとして抽出できるわけではないが，全体の 63.3%で不具合誘発スキーマを全て抽出でき，また全体の 95.6%で少なくとも部分的に抽出できる．この精度は，システムの入力規模が大きいほど，また不具合誘発スキーマの個数が少ないほど高くなる傾向にある．

1.3.3 入力値限局における実行順序に依存したテスト結果への拡張

まず，非決定的なテスト結果が得られる原因について調べられた研究を調査した．Luo らは 51 件の Apache オープンソースプロジェクトから非決定的な不具合を含む 201 のテストに関するコミットを調査し，非決定性の要因を 10 種類に分類した [42]．更に，その内の約 12%が実行順序に依存するものであり，これら 10 種類の非決定的要因の中で大きな割合を占めていることが報告されている．

こうした原因に着目し，実行順序に依存して不具合を誘発する要因の特定に焦点を絞った新しい入力値限局手法である F-CODE (FIL Considered Order DEpendency) を提案する．本手法は最初に，非決定的なテストケースが常に欠陥のある動作を検出するために必要なテストケースの順序を特定する．特定されたテスト実行順序を用いてテスト結果の非決定性を排除することで，OFOT 法などの既存の入力値限局手法の適用が可能となり，不具合誘発要因を特定できるようになる．加えて本手法は，順序依存性の原因となる要因を合わせて特定する．これは，そのパラメータ値を含むテストケースを事前に実行することで，順序依存性を持つテストケースを常に失敗させるようなものを指す．こうした依存関係を特定することで不具合発生を再現させるために完全な情報が入手できるため，より効果的な不具合修正への情報提供が期待できる．

実際のソフトウェアシステムに基づく人工的なテスト結果を用いて F-CODE を評価し，次の結果が示された．

- F-CODE は，不具合を誘発する単一の要因を常に特定できる．一方，同じ対象への単純な OFOT 法の特定成功率は 10 %未満である．
- F-CODE は，テスト結果の非決定性を排除するために，従来の FIL 手法よりも多くのテストケースを追加実行する必要がある． ただし，実行回数の増加割合は，評価に使用したどのシステムでも平均で 2 倍未満である．
- 不具合を誘発する要因が複数存在する場合，F-CODE はそれらを常に特定できるとは限らない．ただし，2 つおよび 3 つの独立した誘発要因が存在する場合に対しそれぞれ 92.6 %および 79.9 %と高い特定成功率を維持する．

1.3.4 OSS を対象とした Java テストコードの再利用可能性分析

まず，Java テストコードの自動生成に焦点を当て，既存のプロジェクトからテストコードを再利用することによるテストコード自動生成の実現を目指し，テストコードを自動的に再利用する方法について構想した．ここでは，テストを実現する最小単位であるテストメソッドごとに一つの移植可能なテストコードを生成する．また，テストメソッド中の識別子を変更する操作のみによって，移植先でも依存関係が保

たれ実行可能となるようなテストメソッドのみを移植の対象とする．そのため，こうした識別子の置換が可能となるための条件について議論する．この条件は，テストメソッドからテスト対象となるコードへの依存に含まれる型情報に注目し，移植元および移植先でその関係性が一致することで識別子を置換しても実行可能となることに基づいている．

さらに，こうした既存テストコードの再利用によるテストコード自動生成手法の実現可能性を探るため，再利用の素材となる既存のテストコードを調査した．調査対象にはGitHub上に存在するテストコードを持つJavaオープンソースソフトウェア（以下，OSS）プロジェクトを用いた．これらの有用性および手法の実現可能性を調査するために以下に示す3つの研究設問を設定し，それぞれの結果を得た．

RQ5 テストコードの移植元として利用できるOSSはどの程度存在するか？

結果：JUnitまたはTestNGを使用するテストコードを含み，一定の品質基準をクリアしたGitHubリポジトリは全体で1,862件存在した．これらは合計約40万ファイルのテストコードを含む．またドメインで分類すると，多いもので約300件のリポジトリが同ドメインに属している．

RQ6 テストメソッドは他の参照先にどれほど依存しているか？

結果：テストメソッドは，その実行に伴ってテストのための補助的な呼び出しを平均1.8種類実行するため，それらを共に移植する必要がある．また平均4.0種類の呼び出しを通してプロダクトコードに依存しており，移植可能であるためにはそうした依存を移植先でも保つ必要がある．

RQ7 既存のテストコードには，テスト対象への依存関係が生成対象であるテストメソッドと共通するテストメソッドはどれほど存在するか？

結果：99件の同ドメインのOSSからの移植を仮定した場合，移植先リポジトリが本来必要とするテストメソッドのうち平均83%は，事前に定義した移植条件を満たすものが移植元に一つ以上存在するため，移植することで同様のテストメソッドを生成できる可能性がある．一方，残りの平均17%では，移植による生成がそもそも不可能である．

1.3.5 再利用に基づく Java テストコード自動生成手法の提案

第 1.3.4 節で述べたテストメソッドの自動移植手法の構想に基づき，実際にテストコードを自動的に生成するためのモデルを提案した．本モデルは，静的構文解析によって移植元および移植先のプロジェクトから必要な情報を抽出する Pick-up モジュール，テストメソッドの依存先の型情報一致判定および識別子の意味的類似判定によりテストメソッド中の識別子の置換候補を推薦する Connect モジュール，テンプレートを用いてテストコードを生成する Rewrite モジュールの 3 つで構成される．

また，このモデルをいくつかの Android アプリケーションに適用した事例調査によって，次の結果が示された．

- 10 の移植先プロジェクトに 217 の移植元プロジェクトからテストメソッドの移植を行うと，平均 953.6 個のテストコードが生成された．
- 生成されたテストメソッドのクラスカバレッジは平均 47.8%となった．また本来のテストコードが依存するクラスに限定したクラスカバレッジは平均 39.4%となった．
- 生成されたテストコードの平均 93.8%がプロダクトコードに対して 1 種類の依存しか持たず，3 種類以上の依存を持つものは平均 1.1%しかない．

1.4 論文の概要

本論文における以後の構成を次に示す．

第 2 章では，本論文で扱う入力値限局およびテストコード生成における背景知識および先行研究について概説する．

第 3 章では，不具合誘発入力値組限局においてロジスティック回帰分析を利用した疑惑値の導出手法を提案する．また実際のシステムの入力モデルに基づく評価によって既存手法との比較を行う．

第 4 章では，ロジスティック回帰分析を使用した不具合誘発入力値組の部分集合の推定による，不審な入力値組を効率的に限局する手法を提案する．また同様に，既存手法および第三章で提案した手法との比較を行う．

第5章では，既存の不具合誘発入力値組限局手法を拡張することによって，実行順序に依存する非決定的なテスト結果にも対応できる手法を提案する．これも同様に，実際のシステムの入力モデルに基づく評価によって既存手法との比較を行う．

第6章では，既存のJavaテストコードを再利用することでテストコードを自動的に生成する手法の構想について紹介し，それに基づいてオープンソースソフトウェアに存在するJavaテストコードの移植可能性に関する調査を行う．

第7章では，第6章での構想に基づいたJavaテストコードの自動生成モデルを提案し，Androidアプリケーションのテストコードを対象に実際に再利用を行う．

最後に，第8章で本論文のまとめを行い，今後の課題について述べる．

第 2 章

背景知識

2.1　入力値限局に関する背景

2.1.1　組み合わせテスト

組み合わせテストとは，複数の入力パラメータ値の組み合わせに焦点を当てたにブラックボックステストの一種である [38,49]．組み合わせテストを行う主な目的は，そうした複数の入力パラメータ値の相互作用による障害発生を効率的に発見することである．この目的のため，組み合わせテスト手法は一定の数以下の入力パラメータ値の組み合わせを全てカバーしつつ，できるだけ少数のテストケースからなるテストスイートを設計する．このように設計されたテストスイートはカバリングアレイとも呼ばれる．ある調査 [37] では不具合を誘発する入力値の組み合わせ数のピーク上限は 4 から 6 と報告されているため，少数の組み合わせに重点を当てる組み合わせテストは実用的であると言える．

また，t 個以下の入力パラメータ値を網羅したカバリングアレイによって t 個以下の入力値の相互作用による障害発生を漏れなく検出することを試みるテストを t-way

テストと呼ぶ．t-way テストでは，最小限のテストケース数で t 個以下のパラメータ値のすべての組み合わせが少なくとも一回テストされる．$t = 2$ の場合の t-way テストはペアワイズテストとしても知られている．t-way テストの t の値を本論文では網羅次数と呼称する．いくつかの t-way テストスイート生成ツールの実装が提案されている [36]．例えば，Czerwonka によって提案された PICT [11, 22] は貪欲なヒューリスティックアルゴリズムを使用している．

以下では，組み合わせテストに関係するいくつかの事項を形式的に定義する．まず，組み合わせテストのテスト対象システム（SUT）の入力モデルは，パラメータ，それらの取り得る値，およびパラメータ間に存在する制約からモデル化される．

定義 1（SUT）

組み合わせテストのテスト対象システム（SUT）の入力モデルは，$\langle P, V, \phi \rangle$ からなる．P はパラメータ $p_1, \ldots, p_n$ の集合である．ここで $n = |P|$ である．V はパラメータ p_i に割り当て可能な値の集合 $V_i (1 \leq i \leq n)$ の集合である．ϕ はパラメータの値の組み合わせに対する制約の集合である．

つまりテストケースとは，制約に違反しない値を各パラメータに割り当てたタプルである．一般的に，制約が多く存在するとテストケース設計の自由度が下がるため，全ての組み合わせを網羅するのに最小となるテストケースの数は増加する．

定義 2（テストケースとテストスイート）

$\langle P, V, \phi \rangle$ が与えられたとすると，テストケースとは $n = |P|$ かつ $v_i \in V_i \, (1 \leq i \leq n)$ であるような，ϕ に違反しない n タプル $(v_1, \ldots, v_n)$ である．またテストスイートはテストケースの集合である．

次にスキーマとは，パラメータの値の組み合わせやパラメータの値の間の相互作用を形式的に表現したものである．特に 1 次スキーマは単一のパラメータ値を指す．この定義は以前の研究 [48] で元々定義されており，最近の関連研究 [50] でも同様に用いられている．

表 2.1: システムの SUT モデル例

パラメータ	値
CPU (= p_1)	Intel (=1), AMD (=2)
Network (= p_2)	Wifi (=1), LAN (=2)
DBMS (= p_3)	MySQL (=1), Sybase (=2)
OS (= p_4)	Win (=1), Linux (=2), Mac (=3)
Browser (= p_5)	IE (=1), Firefox (=2), Chrome (=3)
制約：	
(OS = Mac) → (CPU ≠ AMD)	
(Browser = IE) → (OS = Win)	

定義 3（スキーマ）

一部の k 個のパラメータに固定値が割り当てられ，その他の無関係なパラメータ（-）がある場合，n タプル $(-, v_{n_1}, \ldots, v_{n_k}, ...)$ は k 次スキーマ $(0 < k \leq n)$ であるという．$k = n$ の場合，テストケース自体は k 次スキーマである．

またスキーマ同士の包含関係を次のように定義する．

定義 4（サブスキーマ，スーパースキーマ）

c_l を l 次スキーマ，c_m を m 次スキーマとする（$l < m$）．c_l のどのパラメータも c_m に存在している場合，c_m は c_l を含んでいる．この場合，c_l は c_m のサブスキーマであり，c_m は c_l のスーパースキーマであるといい，$c_l \prec c_m$ と表す．

例として，2 次スキーマ (-, 4, 4, -) は 3 次スキーマ (-, 4, 4, 5) のサブスキーマであり，(-, 4, 4, -) ≺ (-, 4, 4, 5) である．またあるテストケースを表すスキーマのサブスキーマとなるスキーマは，テストケースに含まれていると表現する．

組み合わせテストの実例

表 2.1 は SUT モデルの例を示している．このシステムには CPU，ネットワーク，DBMS，OS，ブラウザの 5 つの設定可能なパラメータがある．最初の 3 つのパラメータには 2 つのとり得る値があり，残り 2 つのパラメータには 3 つのとり得る値

表 2.2: 適用例における 3-way テストスイートおよびテスト結果

テストケース	p_1	p_2	p_3	p_4	p_5	結果
#1	2	2	1	2	3	合格
#2	1	2	2	1	1	合格
#3	2	1	1	1	2	合格
#4	1	2	2	3	2	合格
#5	2	1	2	1	1	合格
#6	1	1	2	3	3	合格
#7	2	2	2	1	2	合格
#8	1	2	1	1	1	失敗
#9	1	1	1	3	2	合格
#10	1	2	1	3	3	合格
#11	1	1	2	2	2	合格
#12	1	2	1	2	3	合格
#13	2	2	2	1	3	合格
#14	1	1	1	1	3	合格
#15	2	1	2	2	3	失敗
#16	1	1	1	1	1	失敗
#17	1	2	1	1	2	合格
#18	2	1	1	2	2	失敗
#19	2	2	2	2	2	合格
#20	2	2	1	1	1	失敗

がある．また，一部のパラメータ値間には制約が存在している．この例では，OS = Mac の場合は必ず CPU ≠ AMD，および Browser = IE の場合は必ず OS = Win という制約がある．つまり，パラメーター値（Mac，AMD），（Linux，IE），および（Mac，IE）の組み合わせは許可されていない．

また表 2.1 に示した SUT の 3-way テストを実現するカバリングアレイを表 2.2 に示す（「結果」列は含まない）．このテストスイートは 20 個のテストケースから構成され，制約を満たしつつ全ての可能な 115 個の 3 次スキーマ (Intel, Wifi, MySQL, -, -), . . ., (-, -, Sybase, Mac, Chrome) を少なくとも一度はテストケースに含んでいる．一方，制約を踏まえたこれらのパラメータ値間のすべての可能な組み合わせは $(2\times2\times2\times3\times3) - 18 = 54$ 通りあり，この全てを網羅するには合計 54 のテストケースが必要となる．このように，組み合わせテストを使用することで網羅基準とのトレードオフとしてテストケースの数を大幅に減らすことができる．

2.1.2 入力値限局

入力値限局は，得られた組み合わせテストの結果から不具合を含む動作を誘発する条件となるような入力パラメータ値の組み合わせを識別する工程である．開発者は組み合わせテストの結果をもとに入力値限局を行なって不具合動作を再現する入力値の条件を特定することで，欠陥のあるコンポーネントを簡単に特定し修復できると考えられる．したがって入力値限局の目的は，所与の組み合わせテストの結果から不具合誘発スキーマを特定することである．

定義 5（不具合誘発スキーマ）

あるスキーマ s を含む全てのテストケースが特定の障害を必ず引き起こし，また s のサブスキーマのどれもが直接障害を引き起こさない場合，スキーマ s を最小次数不具合誘発要因スキーマ（Minimal Failure-inducing Schema）と呼ぶ．本論文では最小次数不具合誘発要因スキーマを単に不具合誘発スキーマと呼称する．

また後述するように，一部の入力値限局手法ではテストスイート全体の結果から不具合誘発スキーマの可能性がある全てのスキーマを先に抽出し，そこから不具合誘発スキーマを絞り込む方法を取るものがある．こうした不具合誘発スキーマの可

表 2.3: 適用例における候補スキーマの一覧

番号	候補スキーマ
s_1	(1, -, -, 2, -)
s_2	(1, 1, -, 2, -)
s_3	(1, 2, -, 2, -)
s_4	(1, -, 1, 2, -)
s_5	(1, -, 2, 2, -)
s_6	(1, -, -, 2, 2)
s_7	(1, -, -, 2, 3)
s_8	(2, 2, -, -, 1)
s_9	(2, -, 1, -, 1)
s_{10}	(-, 1, 2, -, 2)
s_{11}	(-, 2, 1, -, 1)

能性があるスキーマを次のように定義する．

定義 6（候補スキーマ）

テストスイート T とテストオラクル $R : tc \subseteq T \rightarrow \{pass, fail\}$ が与えられたとき，$tc \supseteq s$ であるようなテストケース tc が T に含まれ，かつ，全ての $tc \supseteq s$ であるようなテストケース tc が $R(tc) = fail$ となるようなスキーマ s を候補スキーマと呼ぶ．

不具合誘発スキーマおよび候補スキーマの実例

表 2.1 で表されたシステムにおいて，OS = Mac かつ Browser = Firefox の場合，障害が発生するとする．また，CPU = AMD かつ Network = Wifi かつ DBMS = Sybase の場合にも障害が発生するとする．したがって，2 次スキーマ (1, -, -, 2, -) と 3 次スキーマ (-, 2, 1, -, 1) の 2 つのスキーマは不具合誘発スキーマである．このとき表 2.2 のテストを実行すると，不具合誘発スキーマを含む 4 つのテストケース#8，#11，#12，および#20 は失敗する．したがって表 2.2 の「結果」列のようなテスト結果が

得られる．候補スキーマの定義に従ってこのテスト結果から 3 次スキーマ以下の候補スキーマを抽出すると，表 2.3 に示される 11 個の候補スキーマが存在することが分かる．下線部の引かれた候補スキーマはそれが実際に不具合誘発スキーマであることを表しており，不具合誘発スキーマは候補スキーマに必ず含まれることが確認できる．後の説明のために，各候補スキーマに番号 s_1 から s_{11} を振っておく．

ここで，s_2 から s_7 の 6 つのスキーマは s_1 のスーパースキーマである．s_1 が不具合誘発スキーマである場合，s_2 から s_7 はサブスキーマに不具合誘発スキーマを含むため，最小次数不具合誘発要因スキーマではないという意味でこれらは不具合誘発スキーマではない．しかし s_1 が不具合誘発スキーマではない場合には s_1 を含むこれらのいずれかが不具合誘発スキーマである可能性があるため，他の候補スキーマをサブスキーマに持つ候補スキーマも抽出しておく必要がある．

2.1.3 入力値限局手法に関する先行研究

これまでにいくつかの入力値組限局手法が提案されている．これらは，テストの実行後にその状況に適応した新たなテストケースの追加生成を伴うか否かによって適応的手法あるいは非適応的手法に分類できる．既存の入力値限局手法をまとめた最近のサーベイ研究には [32] が存在する．

非適応的手法では主に，カバリングアレイ（Covering Array）の設計時に一定の入力値限局能力を有する Locating Array を生成する．例えば，Colbourn と McClary は (d, t)-Locating Array を提案した [19]．これはサイズ $(d + t)$ のスキーマを網羅したカバリングアレイを設計することにより，t 次以下の d 個の不具合誘発スキーマを一意に特定することが可能になる．類似した数学的オブジェクトとして，Martínez らは Error Locating Array を提案した [44]．こちらは不具合誘発スキーマの個数ではなく，不具合誘発スキーマに関係する入力パラメータの構造を仮定して設計を行う．例えば，それぞれの不具合誘発スキーマは 2 つ以下のパラメータから構成され，またそれぞれのパラメータは 2 つ以下の値を取り得る，という仮定を置く．そのほか，Locating Array に関するいくつかの発展形も提案されている．例えば，Hagar らは対象のソフトウェアにおける既知の安全な入力値を設計に使用する部分カバリングアレイ手法を提案した [30]．Nagamoto らはペアワイズテストに焦点を絞り，

所与のテストスイートから (1,2)-Locating Array を効果的に生成する方法を提案した [47]．また Jin らは，制約のある入力モデルに対処できるよう Locating Array を拡張した Constrained Locating Array を提案した [33]．こうした Locating Array の研究とその応用に関する最近の調査は [20] にもまとめられている．

非適応的手法を用いる利点は，テストスイートの設計および実行と，その後の入力値限局のプロセスを完全に分離できる点にある．一方で欠点としては次の通りである．Locating Array を設計するためには，不具合誘発スキーマの次数および個数などの情報が既知である必要がある．あるいは，それらを仮定した上で設計を行い，仮定が適切だった場合にのみ限局が成功することになる．また，Locating Array を構成するテストケースの数は素朴なカバリングアレイに比べて大幅に増加し，高い実行コストが必要となる．こうした制限は実用のうえであまり魅力的ではなく，また現段階ではこれらの根本的な制限は解決されていない．

次に，適応的手法では主に，テストケースの実行結果を元に新たなテストケースを設計し実行することで不具合誘発スキーマを一意に限局する [39,48,62,65,67]．いくつかの手法では，テストに失敗した単一のテストケースを入力として使用する．そしてそのテストケースを構成するパラメータのうち一つの値を変更した追加のテストケースを生成して実行し，その実行結果の変化によって元のテストケースに含まれる不具合誘発スキーマを特定する．こうした手法の多くは不具合誘発スキーマの個数が 1 つの場合にのみ動作することが知られている．例えば，OFOT 法 [48] や Delta Debugging を利用した手法 [39,62,65] では，不具合誘発スキーマの数が 1 つでなければ多くの場合に正しい結果を得ることができない．OFOT 法については 5.1.1 節で詳細に紹介する．こうした問題に対し，Zhang らは複数の不具合誘発スキーマの限局に一部対応した FIC を提案した [67]．また，Niu らは単一のテストケース設計および実行と入力値限局のプロセスを動的に連続動作させることでより効果的に限局を行う ICT を提案した [50]．ICT ではさらに，特定した不具合誘発スキーマが本当にテストを失敗に導いているのかを OFOT 法を拡張したチェッキング機構を用いてチェックすることで，不具合誘発入力値組が複数存在する場合に起きる誤った特定を防ぐことができる．テストに失敗した単一のテストケースを用いるこれらの手法は，障害が発生するとそこでテストケースの順次実行を中断して入力値限局に移

り，得られた限局結果を用いて欠陥箇所を修正したのち，再びテストを行うといったサイクルを前提としている．

また一方で，設計されたカバリングアレイのテストケースを全て実行した後，それら全ての実行結果から入力値限局を行う手法も存在している．昨今の継続的インテグレーション等による自動テスト技術の発展により，カバリングアレイ全体のテスト結果を得ることは十分現実的である．最も原始的な手法として，Yilmaz らは分類木を使って不具合誘発スキーマを推定する方法を提案した [63]．この推定には追加のテストケースを必要としないものの，正確さは保証されない．またこのアプローチの有効性はカバリングアレイの特性に大きく依存し，例えば少数のテストケースからなるカバリングアレイの大部分が失敗した場合にはうまく作用しないことが知られている．Fouche と Shakya らは Yilmaz らの研究にいくつかの改善を加えた発展形を提案している [23,56]．Yilmaz らはまた，推定結果をテストケース生成にフィードバックするフレームワークも考案している [64]．

別の手法では，まず不具合誘発スキーマである可能性のあるすべての候補スキーマを抽出し [27-28]，それらが実際に不具合誘発入力値組であることを確かめる追加のテストケースを生成する方法を取る [26, 27, 51, 55, 59, 62, 66, 68]．Shi らによる AIFL は単一の不具合誘発スキーマを想定し [59]，また Wang らによって拡張された InterAIFL は複数の不具合誘発スキーマを扱うことができる [62]．Zheng らによって提案された comFIL は消去法によって複数の不具合誘発スキーマを発見でき，また全候補スキーマから一度のテストで最も多く候補を削減できるようなテストケースを生成する [68]．また Niu らの手法では，タプル関係木（Tuple Relationship Tree）を構築して各候補スキーマ同士の関係を記述することで追加テストケース設計の最適化を試みている [51]．また Gandihari ら [30-31] の提案した BEN では，算出された疑惑値に基づいて候補スキーマを順位付けることで効率的な入力値限局を行う [26,27,55]．いくつかの研究では，BEN は入力値限局手法の有力な候補として扱われている [17,25]．BEN については 3.1.1 節で詳細に紹介する．

第 3 章で提案する FROGa は既存の適応的手法である BEN を拡張したものであり，また第 5 章で提案する F-CODE も既存の適応的手法である OFOT 法を拡張したものである．したがってこれらは適応的手法に属している．一方で第 4 章で提案す

```java
import static org.junit.jupiter.api.Assertions.assertEquals;

import example.util.Calculator;

import org.junit.jupiter.api.Test;

class MyFirstJUnitJupiterTests {

    private final Calculator calculator = new Calculator();

    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }

}
```

図 2.1: 単純なテストコード例（a first test case）

る FROGb は非常に不審な候補スキーマのみの抽出に焦点を当てるため，追加テストケースの実行を必ずしも前提とはしていない．

2.2 テストコードの自動生成と再利用に関する背景

2.2.1 Java テストコードの構造

一般的に，テストフレームワークを用いてテストコードを作成できる．Java 用の単体テストフレームワークとして，JUnit [7] と TestNG [13] がよく知られている．

テストコードでは，作成されたテストメソッドを各テストフレームワークが識別して実行することで自動的にテストが行われる．このとき，呼び出したテストメソッドが何らかの例外を返すことで，テストの成否を判断できる．意図しない実行時エラーによる例外を検出するほか，特定の条件下でアサーション例外を発生させるメソッドを使用することでテストを行うことができる．例えば，JUnit における assertEquals メソッドは 2 つ以上の引数を要求し，それらが一致しない場合にアサーションを発生させる．これにより，実際の出力と期待結果を比較することで正しい結果が得られていることを確認できる．

単純な単体テストの例として，JUnit5 の公式ドキュメントに掲載されている a first test case [8] と題されたテストコードスニペットを図 2.2.1 に示す．ここでは

FirstJUnit5JupiterTest クラス中で addition メソッドが宣言され，これには JUnit ライブラリからインポートした@Test アノテーションが付与されている．これにより，addition がテストメソッドとして認識される．テストメソッド内では，同様に JUnit ライブラリからインポートされた assertEquals メソッドが呼び出され，2 および calculator.add(1, 1) が引数となっている．これらは等しくなるためアサーションが発生せず，テスト結果は常に合格となる．

Eclipse などの統合開発環境や Maven などのビルドツールでは，プロジェクトのビルド時にテストメソッドを実行するよう設定することができる．これにより，プロジェクトに変更を加えて再ビルドを行う際に自動的にテストを実施できる．またテストコードが必要なサードパーティ製ライブラリを設定ファイルに記述しておくことで自動的にそれらを入手して使用することも可能である．さらに継続的インテグレーションを使用することで，変更のコミット時など任意のタイミングで自動的にテストを実施できる．

Java プロジェクトの一般的なディレクトリ構造として，ソースコードファイルを格納する src ディレクトリの直下に main ディレクトリおよび test ディレクトリが配置され，test ディレクトリ下にテストコードが配置される．これを根拠として，本研究では test ディレクトリ下にある Java ソースコードをテストコードとみなす．また main ディレクトリ下にはそのプロジェクトが提供する機能を実現する Java ソースコードファイルが存在する．本研究ではこれらをプロダクトコードと呼称する．テストコードは通常，メソッド呼び出しやインスタンス生成などを通じてテスト対象となるプロダクトコードへの依存を有している．

2.2.2 テストコードの自動生成に関する先行研究

テストコードの自動生成を目的としたいくつかの研究が行われている．EvoSuite はいくつかの選択可能な基準で最適化された関数単位での Java テストコードを自動的に生成することができる [14,24]．しかし，テストコードの質が低いことが指摘されている [57]．また，ごく簡単な単体テストにしか対応していない．関数単位での回帰テスト用のコードを自動生成する Randoop [53] も同様の問題を抱えている．他にも，JavaDoc などの構造化された自然言語で書かれた仕様書からテストコードを自

動生成する手法 [28,46,61] や，振る舞い駆動開発における製品の振る舞いを独自の記法で記述したファイルからテストコードを自動生成する手法 [2,6,40] がいくつか提案されている．しかし，これらの手法はテストコードの原型となる文書の準備を要するため，純粋な自動生成とは言い難い．また，深層学習等によるプログラム合成手法は，研究用に定義されたドメイン固有言語を対象としているものが多く，あるいはコード生成の精度が十分でないなど，実用的なテストコード生成には至っていない [15]．このように，結合テストなどの複雑なテストコードを高精度かつ手軽に自動生成する手法は未だ実現されていない．

一方，既存のコードを再利用することで開発効率や信頼性が向上することが知られている [35]．したがって，既存のテストコードを再利用するアプローチにも一定の効果が期待できる．テストコードではないが実際に，既存のソースコード片を再利用することでコード生成を実現するいくつかの方法が提案されている [31,41]．また Sondhi ら [60] は，数種類の Java および Python の OSS のテストコードの再利用性について調査し，類似した機能へのテストコードが存在することや，テストコードを移植することで同様のテストが行える可能性を示した．ただし，あるライブラリのテストコードを別のライブラリに移植する際には，内容の書き換えを手動により行っている．このように，テストコードを自動的に移植する手段については未だ開拓されていない．

第 3 章

入力値限局におけるロジスティック回帰を用いた分析精度の改善

3.1 周辺知識

3.1.1 BEN

代表的な入力値限局手法として，Ghandehari らは BEN を提案している [27]．この手法では，テストスイート全体のテスト結果から候補スキーマを抽出したのち，それらの疑わしさを算出して順位付けを行い，順位の高いものから順に新たなテストケースを生成し実行することによって不具合誘発スキーマを特定する．新たなテストケースの生成は，他に一切の候補スキーマを含まないようなテストケースを作成する AIFL [59] と呼ばれる既存手法を踏襲している．この手法の特徴は，候補スキーマの疑わしさを分析し順位付けを行うことで実行するテストケースを効率化する点に

ある．この疑わしさの順位付けは次のように行われる．これらの式は，Ghandehari らのヒューリスティックに基づいて独自に作成されたものである．

まず，候補スキーマ c について $\rho_c(c)$ と $\rho_e(c)$ の値を求める．次に，全候補スキーマのうち $\rho_c(c)$ と $\rho_e(c)$ の大きさの順位を求め，その順位を足した数 R を求める．最後に，全候補スキーマを R の小ささによって順位付ける．$\rho_c(c)$ および $\rho_e(c)$ は次のように求められる．

$$\rho_c(c) = \frac{1}{|c|} \sum_{\forall o \in c} \rho(o)$$

$$\rho_e(c) = Min\left(\frac{1}{|c|} \sum_{o \in f \cap o \notin c} \rho(o), \forall f \in F \right)$$

ここで，o は c に含まれるパラメータ値であり，f はテストスイート F に含まれるテストケースである．また $\rho(o)$ は次のように求められる．

$$\rho_c(c) = \frac{1}{3}\left(u(o) + v(o) + w(o)\right)$$

$$u(o) = \frac{\{f \in F_i | r(f) = fail \cap o \in f\}}{|\{f \in F_i | r(f) = fail\}|}$$

$$v(o) = \frac{\{f \in F_i | r(f) = fail \cap o \in f\}}{|\{f \in F_i | o \in f\}|}$$

$$w(o) = \frac{\{c | o \in c \cap c \in \pi\}}{|\pi|}$$

ここで，$r(f)$ は f のテスト結果 $r \rightarrow \{pass, fail\}$ であり，π は全候補スキーマの集合である．

3.1.2 ロジスティック回帰分析

ロジスティック回帰は二項従属変数の回帰に対するよく知られた統計モデルであり，教師あり機械学習手法として主に用いられる [21]．ソフトウェア工学分野における

例としては，各種ソフトウェアメトリクスを独立変数として，また欠陥の有無を従属変数として使用して学習を行うことで，故障しやすいソフトウェアコンポーネントを推定する用途で利用されている [16,18,34].

ロジスティック回帰モデルは次の式に基づいている．

$$Pr(Y = 1|x_1, \cdots, x_n) = \frac{1}{1 + e^{-(b_0 + b_1 x_1 + \cdots + b_n x_n)}}$$

ここで，$x_1, \ldots, x_n$ は独立変数，$b_1, \ldots, b_n$ はその係数，b_0 は切片，Y は 0 か 1 の値を取る二項従属変数，Pr は $x_1, \ldots, x_n$ の値が与えられた場合に $Y = 1$ となる条件付き確率である．ロジスティック回帰では，所定のアルゴリズムによって正解ラベル付きの学習データから最も条件付き確率と正解値の誤差が小さくなるような $b_1, \ldots, b_n$ の値を回帰係数として求め，回帰式を得る．これを分類器として使用する場合は回帰式にテストデータを代入して二項従属変数が正値となる条件付き確率を算出し，一般的に 0.5 に設定される閾値によってその分類結果を提示する．

一方で，ロジスティック回帰はデータ分析手法にも利用されており，本研究では分析手法としてのロジスティック回帰分析を使用する．前述のように，ロジスティック回帰では学習データから回帰曲線を学習することによって独立変数 $x_1, \ldots, x_n$ それぞれの回帰係数 $b_1, \ldots, b_n$ を求めることができる．ロジスティック回帰分析は，これらの回帰係数をそれぞれの独立変数の単位増加による確率 Pr への影響の程度と見なすことができることを利用している [54]．係数が大きい場合，その独立変数の値の変化が確率 Pr の変動に大きな影響を与えていることを示す．したがって回帰係数は，二項従属変数が正値であることに対してその独立変数が持つ寄与度として読み取ることができる．

3.2 提案モデル：FROGa

3.2.1 概念構成

本節では，ロジスティック回帰分析を用いて従来の入力値限局手法の精度を改善させる手法として FROGa を提案する．本手法がベースとする既存手法である BEN は独自の計算式によって候補スキーマの疑わしさを分析し順位付けた．一方，提案モ

デルではロジスティック回帰分析によって得られる回帰係数を疑惑値として使用することで，より精度の高い順位付けを試みる．

ロジスティック回帰分析によって疑惑値を算出できる理屈を次に述べる．先に述べたように，回帰係数はその独立変数の単位増加が二項独立変数が 1 となる確率への影響の程度と見なすことができる．ここで，独立変数をある候補スキーマにおけるあるテストケースとの包含関係を表す二値（1 = 含まれる，0 = 含まれない）とし，従属変数をテスト結果を表す二値（1 = 失敗，0 = 成功）とする．このとき回帰係数は，対応する独立変数が 0 から 1 に変化した場合，すなわち候補スキーマがテストケースに含まれない状態から含まれる状態へと変化した場合に，その変化がテストケースが失敗する確率に与える影響の大きさと見なすことができる．したがってこの影響が高ければ不具合誘発スキーマである可能性が高いため，回帰係数は対応する候補スキーマの疑惑値として機能することが期待される．

3.2.2 モデル仕様

FROGa への入力として，テストスイートとそれぞれのテストケースのテスト結果，および想定する最大のスキーマ次数 k が必要となる．テスト結果は，何らかのテストオラクルによって成功および失敗に分類されていることを前提とする．またスキーマ次数 k は基本的に，入力するテストスイートが網羅する組み合わせ数を指定する．

まず，入力されたテストスイートおよびそのテスト結果から，サイズが k 以下の候補スキーマを全て抽出する．この候補スキーマの集合を S_k とする．

次に，S_k の各候補スキーマと各テストケースの包含関係を表すデータテーブル Φ を作成する．あるテストケースを表すスキーマ st にある候補スキーマ sc が含まれていることを表す関数 $inc(sc, st)$ を次のように定義する．

$$inc(sc, st) = \begin{cases} 1 & (sc \prec st) \\ 0 & (sc \not\prec st). \end{cases}$$

データテーブル Φ は関数 $inc(sc, st)$ を用いて次のように表現できる．

$$\Phi = \begin{bmatrix} inc(sc_1, st_1) & \dots & inc(sc_m, st_1) \\ \vdots & \ddots & \vdots \\ inc(sc_1, st_n) & \dots & inc(sc_m, st_n) \end{bmatrix}$$

ここで m は抽出した候補スキーマの数 $|S_k|$ であり，n はテストスイートを構成するテストケースの数である．すなわち，データテーブル Φ の列が各候補スキーマに，また行が各テストケースに対応しており，Φ はこれらの包含関係をエンコードしたものとなる．

また加えて，各テストケースのテスト結果を表す疑似ブールベクトル R を作成する．テストスイートが n 個のテストケース $st_1, \ldots, st_n$ からなる場合，st のテスト結果を成功 = 0 および失敗 = 1 にエンコードする関数 $result(st) \rightarrow \{0, 1\}$ を用いて，R は次のように表現できる．

$$R = [result(st_1), \ldots, result(st_n)]^T$$

FROGa は次に，Φ の各列ベクトルを独立変数，R を従属変数としてロジスティック回帰を実行する．これにより，各候補スキーマに対応する独立変数の回帰係数を取得できる．

最後に，得られた回帰係数の値を疑惑値として候補スキーマの順位付けを行う．疑惑値が高いほどその候補スキーマが不具合誘発スキーマである可能性が高いと判断できる．

3.2.3 単純化

次にエンコードの単純化について述べる．要約すると，テストに合格したテストケースに対応する Φ および R の行ベクトルを省略できる．

候補スキーマの定義から，候補スキーマは合格したどのテストケースにも含まれていない．よって Φ の行ベクトルのうち，合格したテストケースに対応する行ベクトルは全ての値が 0 であり，また対応する R の値も 0 となる．したがって，合格したテストケースに対応する行ベクトルは全て同一となる．一方ロジスティック回帰では，行ベクトルは 1 サンプルに相当し，全く同一のサンプルを 2 つ以上入力しても 2 つ目以降のサンプルが回帰式を更新することはない．したがって，合格したテストケースに対応する行ベクトルは 1 つのみでも問題ない．これによって FROGa は Φ および R に代えて次のように単純化された Φ' および R' を使用できる．

$$\Phi' = \begin{bmatrix} inc(sc_1, st_1) & \dots & inc(sc_m, st_1) \\ \vdots & \ddots & \vdots \\ inc(sc_1, st_f) & \dots & inc(sc_m, st_f) \\ 0 & \dots & 0 \end{bmatrix}$$

$$R' = [1, \dots, 1, 0]^T$$

ここで F は失敗したテストケースの集合であり，$f = |F|$ である．したがって，Φ' は m 行 $f+1$ 列のデータテーブルであり，R' は $f+1$ 次元の列ベクトルである．合格したテストケースの情報は候補スキーマの抽出に使用されるため，この単純化によって実行するテストケースの数を減らせるわけではない．

3.2.4 適用例

本モデルの理解を深めるため，適用例を示す．表 3.1 は，表 2.2 のテストスイート結果および表 2.3 の候補スキーマから作成された Φ および R を表している．また表 3.2 はその単純化である Φ' および R' を表している．表 3.3 は，Φ' および R' を入力としてロジスティック回帰を行い得られた各候補スキーマの回帰係数，およびそれによる順位を示す．また表 3.3 中における下線部は，その候補スキーマが不具合誘発スキーマであることを示している．この例では，ロジスティック回帰は scikit-learn [12] で実装されたものを使用した．またロジスティック回帰のパラメータは全てデフォルトのものを使用した．この例では不具合誘発スキーマが得られたランキングの頂点に位置しており，FROGa がうまく機能していることが確認できる．

表 3.1: 適用例において作成された Φ および R

テストケース	Φ											R
	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	
#1	0	0	0	0	0	0	0	0	0	0	0	0
#2	0	0	0	0	0	0	0	0	0	0	0	0
#3	0	0	0	0	0	0	0	0	0	0	0	0
#4	0	0	0	0	0	0	0	0	0	0	0	0
#5	0	0	0	0	0	0	0	0	0	0	0	0
#6	0	0	0	0	0	0	0	0	0	0	0	0
#7	0	0	0	0	0	0	0	0	0	0	0	0
#8	0	0	0	0	0	0	0	0	0	0	1	1
#9	0	0	0	0	0	0	0	0	0	0	0	0
#10	0	0	0	0	0	0	0	0	0	0	0	0
#11	1	1	0	0	1	1	0	0	0	1	0	1
#12	1	0	1	1	0	0	1	0	0	0	0	1
#13	0	0	0	0	0	0	0	0	0	0	0	0
#14	0	0	0	0	0	0	0	0	0	0	0	0
#15	0	0	0	0	0	0	0	0	0	0	0	0
#16	0	0	0	0	0	0	0	0	0	0	0	0
#17	0	0	0	0	0	0	0	0	0	0	0	0
#18	0	0	0	0	0	0	0	0	0	0	0	0
#19	0	0	0	0	0	0	0	0	0	0	0	0
#20	0	0	0	0	0	0	0	1	1	0	1	1

表 3.2: 適用例において作成された Φ' および R'

テストケース	Φ'											R'
	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	
#8	0	0	0	0	0	0	0	0	0	0	1	1
#11	1	1	0	0	1	1	0	0	0	1	0	1
#12	1	0	1	1	0	0	1	0	0	0	0	1
#20	0	0	0	0	0	0	0	1	1	0	1	1
他	0	0	0	0	0	0	0	0	0	0	0	0

表 3.3: 適用例において得られた各候補スキーマの回帰係数およびその順位

番号	候補スキーマ	回帰係数（疑惑値）	順位
s_{11}	(-, 2, 1, -, 1)	1.002809	1
s_1	(1, -, -, 2, -)	0.723898	2
s_8	(2, 2, -, -, 1)	0.402193	3
s_9	(2, -, 1, -, 1)	0.402193	3
s_3	(1, 2, -, 2, -)	0.385049	5
s_4	(1, -, 1, 2, -)	0.385049	5
s_7	(1, -, -, 2, 3)	0.385049	5
s_2	(1, 1, -, 2, -)	0.338850	8
s_5	(1, -, 2, 2, -)	0.338850	8
s_6	(1, -, -, 2, 2)	0.338850	8
s_{10}	(-, 1, 2, -, 2)	0.338850	8

表 3.4: 実験に使用する SUT

SUT	パラメータサイズ	制約サイズ	#2-tuples	#3-tuples	#4-tuples
SystemMgmt	10; $2^5 3^4 5^1$	27; 2^{13}	310	1,982	7,770
Storage3	15; $2^9 3^1 5^3 6^1 8^1$	48; $2^{38} 3^{10}$	1,020	11,840	89,623
ProcessorComm2	25; $2^3 3^{12} 4^8 5^2$	81; $1^4 2^{121}$	2,525	53,228	781,772
Healthcare4	35; $2^{13} 3^{12} 4^6 5^2 6^1 7^1$	48; $2^{38} 3^{10}$	5,707	191,398	4,625,149
SPINS	18; $2^{13} 4^5$	38; 2^{13}	979	12,835	116,332
SPINV	55; $2^{42} 3^2 4^{11}$	133; $2^{47} 3^2$	8,741	369,976	11,427,629

3.3 評価方法

3.3.1 研究設問

FROGa の有効性を評価するために以下の研究設問を設定した．

- RQ1：FROGa は BEN と比較して不具合誘発スキーマとしての疑わしさを順位付けする精度を向上させるか？
- RQ2：FROGa と BEN の処理コストに差はあるか？

またこれらの研究設問に答えるため，提案手法である FROGa および従来手法である BEN を用いて与えられた組み合わせテストの結果から候補スキーマの疑わしさを順位付けし，その精度および処理にかかった時間を比較する．

3.3.2 実験対象

実験対象には実際に報告された不具合に基づく組み合わせテスト結果ではなく，人工的に作成された組み合わせテスト結果を用いる．現実に組み合わせテストによって欠陥検出が報告されたバグレポートはわずかしか存在しないため，こうした少ない実験対象のみの使用では評価結果の妥当性を保証できない．そのため特定の不具合誘発スキーマが誘発する不具合の存在を想定し，その場合のテスト結果を得ることで，多数の実験対象を実験に使用する．現実に発生し得る不具合は我々が想定す

表 3.5: 各 t-way テストスイートを構成するテストケース数

SUT	$t=2$	$t=3$	$t=4$
SystemMgmt	19	58	155
Storage3	55	243	749
ProcessorComm2	34	170	782
Healthcare4	53	321	1,811
SPINS	28	116	420
SPINV	67	332	1,493

る多数の不具合誘発スキーマに包含されるため，人工的に生成されたテスト結果を実験に使用することが妥当性に与える影響は限りなく低いと考えている．

不具合を想定するテスト対象として，6 つの実際のソフトウェア（SUT）を実験に使用した．表 3.4 に各 SUT が持つパラメータと制約のサイズ，および制約を満たす t 次スキーマの数（$t = 2, 3, 4$）を示す．SUT のパラメータサイズは $|P|; g^{k1}g^{k2}\ldots g^{kn}$ として表されている．ここで，g_i 個の値を持つパラメータが k_i 個あり，パラメータの個数は $|P|$ である．また制約のサイズは，各 j に対して l_j 個のリテラルを持つ m 個の変数と h_j 節を持つ連言標準形 $l; h^{l1}h^{l2}\ldots h^{lm}$ として表されている．SystemMgmt, Storage3，ProcessorComm2，および Healthcare4 の 4 つの SUT は，IBM 製品プログラムの特定のバージョンを指す．また SPINS および SPINV は，オープンソースのモデル検査ツールである SPIN におけるシミュレータ（Simulater）および検証ツール（Verifier）を指す．これらの SUT モデルは Cohen らの論文 [58] で公開されているものの中から入力の規模に多様性を持たせられるよう無作為に選択した．

また不具合が想定された SUT をテストするテストスイートを設計するために，Microsoft が提供するよく知られたカバリングアレイ生成ツールである Pict [11] を使用した．Pict は，SUT モデルおよび組み合わせテストが網羅する組み合わせの大きさを入力することでカバリングアレイを生成することができる．

人工的なテスト結果の生成方法を次の 3 ステップに示す．

（1）テストスイートの作成

Pict を用いて各 SUT モデルの t-way テスト（$t = 2, 3, 4$）を実現するカバリン

グアレイを作成した．その結果，6 種類の SUT モデルと 3 通りの *t*-way テストの組み合わせから，18 種類のテストスイートが作成された．表 3.5 は，各テストスイートを構成するテストケースの数を示している．

(2) 不具合誘発スキーマの決定

それぞれのテストスイートにおいて，存在を想定する不具合誘発スキーマのパターンを無作為に決定する．まず，各不具合誘発スキーマの数を 1 から 3 の間で無作為に決定する．また各不具合誘発スキーマの次数を，*t*-way テストを対象としている場合 2 から *t* の間で無作為に決定する．この次数の制限は，全ての不具合誘発スキーマを含むテストケースがは少なくとも一度はテストされることを目的としている．不具合誘発スキーマを構成する入力値は，後述する「制約によってロックされた」特定の入力値を回避しつつ無作為に決定された．以上の反復により，各テストスイート毎に重複なく 10,000 個の不具合誘発スキーマのパターンを用意する．

(3) テスト結果の生成

想定されている不具合誘発スキーマに従って組み合わせテストのテスト結果を求める．ここで，不具合誘発スキーマを一つでも含むテストケースの結果は失敗となり，そうでなければ合格となる．最終的に，各テストスイート毎に組み合わせテスト結果の 10,000 個のサンプルを取得した．ただし，4-way テストのテスト結果については 1,000 個のサンプルのみを取得した．これは実験の時間的な制約による．

ここで「制約によってロックされた」入力値とは，特定の制約の影響によってテストケースに常に同時に含まれる一対の入力値に属する入力値を指す．こうした入力値が不具合誘発スキーマに含まれる場合，意図しない別の不具合誘発スキーマが同時に存在することになり，評価に余計な混乱を招くことになる．実験を簡単にするために，事前に制約を調査し，そのような入力値が不具合誘発スキーマに含まれることを禁止した．

3.3.3 実験環境

BEN および FROGa のアルゴリズムを Python3.5.1 で実装し，MacBook Pro(2017) で実行した．このマシンはプロセッサとして 3.1GHz デュアルコア Intel Core i5，メモリとして 16GB 2,133MHz LPDDR3 の標準的なスペックを持つ．また，FROGa におけるロジスティック回帰の実装には機械学習用のオープンソース Python ライブラリである scikit-learn [12] を使用した．またこのとき，全てのパラメータにデフォルトで設定されているものを使用した．さらに，一つの組み合わせテスト結果に対する実行処理におけるタイムアウト基準は 3,600 秒とした．組み合わせテスト結果の符号化には Φ および R ではなく単純化された Φ' および R' を使用した．

3.3.4 評価指標

RQ1 に答えるために，それぞれの手法で疑わしさを順位づけられた候補スキーマ集合の中で不具合誘発スキーマがどれだけ高い順位に位置しているかを比較する．このために，ランクベースの精度評価指標である MAP および top-k % accuracy を使用する．また RQ2 に答えるために，順位付けにかかった処理時間を測定する．

MAP（Mean Average Precision）は，順位付け能力の精度を評価するランクベースの精度評価指標である [43]．この指標は，検索キーワード等のクエリを与えるとそれに対する関連度が高い順に並んだ検索結果が返されることを期待されるクエリ情報検索システムの評価に多く用いられている．MAP の値は 0 から 1 の値を取り，値が大きいほど順位付けが正確だと判断される．ここでは，入力となる組み合わせテスト結果をクエリとして扱い，順位付けされた候補スキーマのリストを検索結果として扱う．MAP を計算するにはまず，クエリ集合 Q のうちある 1 つのクエリ q における $AveP_q$（Average Prediction）を次の式によって計算する．

$$AveP_q = \frac{1}{|R_q|} \sum_{n=1}^{|R_q|} prec@n(q)$$

ここで $|R_q|$ はクエリ q に実際に関連している検索結果の数であり，ここでは不具合誘発スキーマの個数として扱う．また $prec@n(q)$ は q に対する検索結果の上位 n 個のうち実際に関連している検索結果の割合，すなわち不具合誘発スキーマの割合であ

る．そして，MAP は Q 内の全てのクエリ $q_1, \ldots, q_{|Q|}$ の $AveP_q$ の算術平均から次のように導かれる．

$$MAP = \frac{1}{|Q|} \sum_{i=1}^{|Q|} AveP_{q_i}$$

次に，top-k% accuracy は top-k accuracy に基づいたランクベースの評価指標である．top-k accuracy は，順位付けされた全候補スキーマのうち全ての不具合誘発スキーマが上位 k 内に位置付けられている場合を成功とみなし，そうでない場合を失敗とみなした場合において複数回の試行におけるカットオフ順位 k での成功率を指す．一方，今回の実験では試行ごとに候補スキーマの数が異なるためこの指標を単純に適用できない．そこで，top-k accuracy の代わりに top-k% accuracy を新たに考案し使用する．これは，全候補スキーマの上位 k%（小数点以下切り上げとする）内に全ての不具合誘発スキーマが位置付けられている場合を成功とみなした場合のカットオフ割合 k での成功率を指す．これは次のように定式化される．

$$\text{top}-k\% \text{ accuracy} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} InTopKp(k, q_i)$$

また加えて，次の方法によって不具合誘発スキーマ数の違いによる評価のばらつきを補正する．以下は候補スキーマの個数が 100 の場合の例である．まず，不具合誘発スキーマが 1 個だけ存在し，それが 1 位に位置付けられた場合を考える．次に，不具合誘発スキーマが 3 個存在し，それらが 1 位から 3 位に位置付けられた場合を考える．これらはどちらも最良の結果であるが，前者は $k = 1$ が成功と評価される一方，後者は $k = 1$，$k = 2$ の場合は成功とならず，$k = 3$ の場合に初めて成功と評価されることになり矛盾が発生する．この問題に対処するため，より上位に位置す不具合誘発スキーマの存在を計算時に無視するように順位を変更する．これにより，後者の場合の順位は（1 位，2 位，3 位）から（1 位，1 位，1 位）に変更され，$k = 1$ で成功と評価される．

3.4 評価結果

実験結果を以下に示す．入力として Healthcare4 および SPINV の 4-way テストを使用した場合，タイムアウト基準として定義された 3,600 秒以内に終了しなかった

表 3.6: BEN と FROGa における MAP の比較

	2-way		3-way		4-way	
SUT	BEN	FROGa	BEN	FROGa	BEN	FROGa
SystemMgmt	0.362	0.599	0.327	0.692	0.232	0.759
Storage3	0.515	0.847	0.231	0.853	0.111	0.794
ProcessorComm2	0.407	0.635	0.337	0.797	0.197	0.757
Healthcare4	0.586	0.806	0.297	0.898	NA	NA
SPINS	0.390	0.664	0.088	0.737	0.045	0.756
SPINV	0.521	0.809	0.238	0.698	NA	NA
平均	0.464	0.727	0.253	0.779	0.146	0.767

ため各評価指標値は欠損値（NA）となっている．これについて，時間的コストの理由から全てのサンプルを試行したわけではないが，Healthcare4 と SPINV の両方で無作為に選択された 5 個のサンプルが全て 3,600 秒内に終了しなかったことを確認している．また実行ログを確認したところ，タイムアウト時点でどのサンプルも候補スキーマの抽出処理が完了しておらず，疑わしさの算出まで辿り着けていなかった．

3.4.1 RQ1 に関する実験結果

表 3.6 に BEN と FROGa における MAP の比較を示す．欠損値を除いた 16 種類のテストスイートにおける FROGa の MAP の平均値は 0.757 であり，BEN の MAP の平均値は 0.305 である．結論として，FROGa の MAP が BEN を大きく上回っていることを確認できる．また，図 3.4.1 に両手法における *AveP* の分布の比較をボックスプロットにより示す．上のグラフは各 SUT ごとに比較し，下のグラフは各 *t*-way テストごとに比較している．これらには欠損値である Healthcare4 と SPINV の 4-way テストを対象とした場合の結果は含まれない．SUT の違いによる差はあまり見られない一方，カバリングアレイが網羅するスキーマサイズの違いによる差は顕著である．BEN では網羅度を上げると MAP は減少する傾向にあるが，FROGa は網羅度を上げても高い精度を維持することが分かる．

また，表 3.7 に全サンプルのうち $AveP = 1$ となったサンプルの割合の比較を示

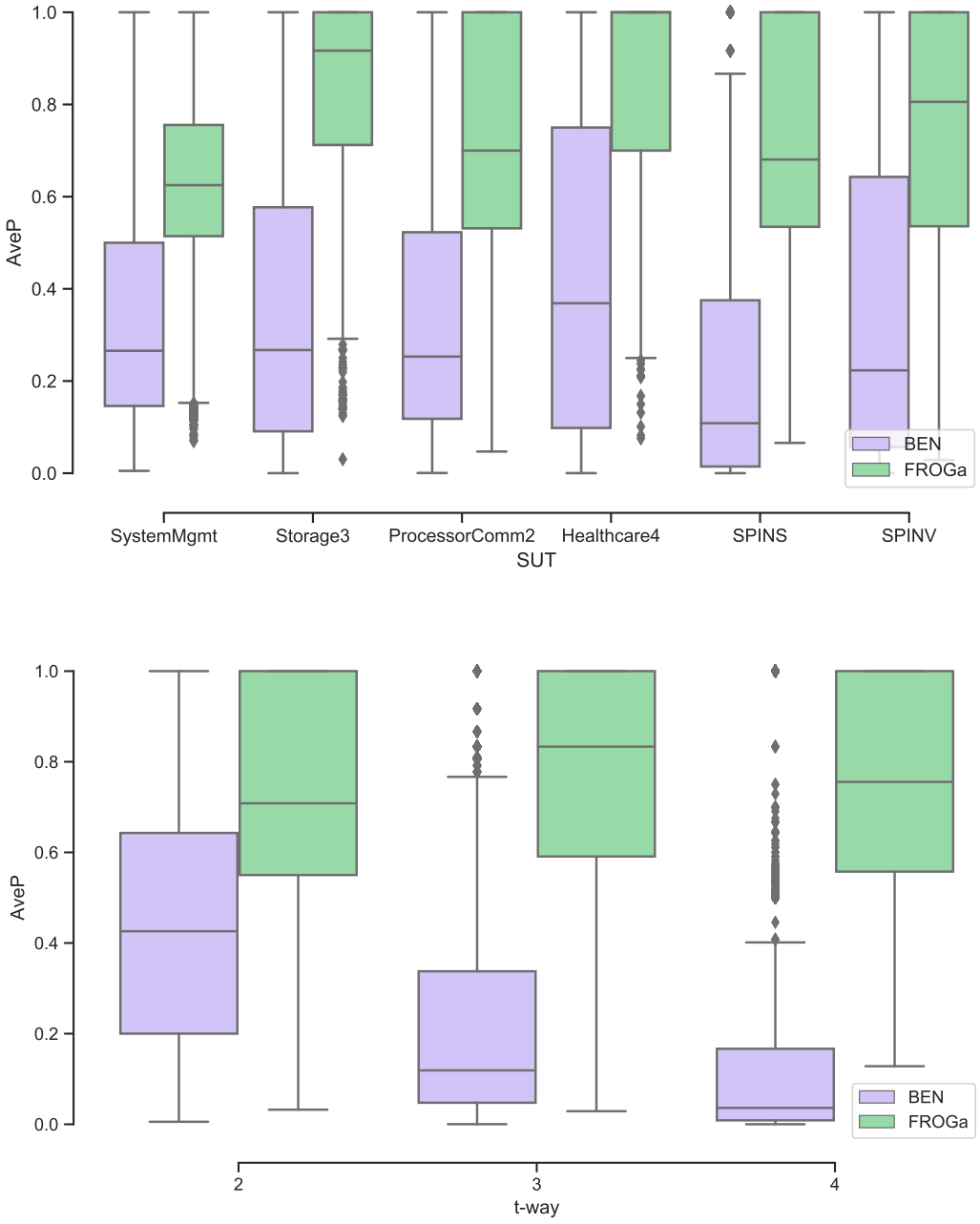

図 3.1: BEN および FROGa における試行毎の適合率 *AveP* の分布比較

表 3.7: BEN と FROGa における *AveP*=1 となった割合の比較

	2-way		3-way		4-way	
SUT	BEN	FROGa	BEN	FROGa	BEN	FROGa
SystemMgmt	3.2%	8.3%	16.5%	43.3%	8.1%	41.5%
Storage3	14.0%	45.1%	5.2%	49.8%	2.3%	45.7%
ProcessorComm2	12.9%	22.7%	14.5%	42.7%	9.3%	42.2%
Healthcare4	26.2%	44.6%	16.1%	61.1%	NA	NA
SPINS	7.5%	19.4%	2.8%	33.7%	6.0%	37.9%
SPINV	22.4%	45.6%	12.3%	40.0%	NA	NA
合計	14.4%	30.9%	11.2%	45.1%	5.1%	41.8%

す．*AveP* = 1 となった場合，その試行で全ての不具合誘発スキーマがランキングの頂点に位置したことを表している．結果として，FROGa が BEN よりも完全な精度での順位付けを行う能力がはるかに高いことを示している．したがって FROGa は高い確率で完全な精度での順位付けが達成されるため，ランキング上位の候補スキーマが実際に不具合を誘発しているかを確認するための追加テストケースの実行を必要とせず，そのまま修正作業に移ることができることを示唆している．

より詳細な分析のため，表 3.8 に同数の MFS を仮定した試験結果から別々に計算された MAP を示す．全ての場合で，FROGa の MAP が BEN を上回っていることが確認できる．また BEN と FROGa の両方で，MFS の数が少ない場合に精度が上昇する傾向があることが分かる．特に不具合誘発スキーマが 1 つだけ存在する場合，BEN とは異なり，FROGa はほぼ全ての場合で不具合誘発スキーマをランキングの頂点に位置づけることに成功した．また本実験において最も難しい状況であると思われる，不具合誘発スキーマが 3 つ存在する場合の 4-way テスト結果を対象とした場合の MAP を比較すると，BEN では 0.071，FROGa では 0.573 となり，精度に大きな差があることが分かる．先行研究 [50] においても 3 種類の適応型手法において不具合誘発スキーマの数が増加することで入力値限局の品質が低下することが報告されており，本結果はこうした傾向を裏付けている．

また表 3.9 に，FROGa を用いることで BEN と比較して *AveP* が上昇したサンプ

表 3.8: 不具合誘発スキーマが n 個存在する試行毎の MAP の比較

		2-way		3-way		4-way	
SUT	n	BEN	FROGa	BEN	FROGa	BEN	FROGa
SystemMgmt	1	0.721	0.992	0.667	0.999	0.406	0.999
	2	0.423	0.674	0.313	0.700	0.155	0.686
	3	0.278	0.500	0.207	0.562	0.096	0.536
Storage3	1	0.722	1.000	0.290	0.999	0.134	1.000
	2	0.540	0.882	0.227	0.845	0.114	0.732
	3	0.444	0.777	0.194	0.755	0.082	0.624
ProcessorComm2	1	0.720	1.000	0.574	1.000	0.366	1.000
	2	0.411	0.633	0.272	0.758	0.129	0.674
	3	0.257	0.465	0.182	0.649	0.068	0.556
Healthcare4	1	0.882	1.000	0.440	1.000	NA	NA
	2	0.591	0.807	0.272	0.879	NA	NA
	3	0.411	0.664	0.183	0.817	NA	NA
SPINS	1	0.665	1.000	0.157	0.997	0.043	1.000
	2	0.428	0.726	0.074	0.730	0.053	0.700
	3	0.295	0.533	0.045	0.569	0.039	0.593
SPINV	1	0.706	1.000	0.378	0.998	NA	NA
	2	0.526	0.814	0.211	0.622	NA	NA
	3	0.364	0.648	0.128	0.485	NA	NA
平均	1	0.726	0.999	0.421	0.999	0.237	1.000
	2	0.486	0.756	0.228	0.756	0.113	0.698
	3	0.341	0.598	0.157	0.640	0.071	0.578

表 3.9: FROGa を用いることで BEN と比較して *AveP* が上昇した試行の割合

SUT	*t*-way	＞ BEN	= BEN	＜ BEN	= BEN =1
SystemMgmt	2	84.4%	4.1%	11.5%	57.8%
	3	77.9%	16.2%	5.1%	99.3%
	4	91.6%	8.1%	0.3%	100.0%
Storage3	2	80.3%	14.1%	5.5%	91.1%
	3	94.0%	5.3%	0.7%	95.7%
	4	97.0%	2.2%	0.8%	100.0%
ProcessorComm2	2	76.2%	13.0%	10.8%	93.0%
	3	83.7%	14.6%	1.7%	97.6%
	4	90.3%	9.3%	0.4%	100.0%
Healthcare4	2	65.0%	25.7%	9.3%	96.5%
	3	83.0%	16.0%	1.0%	99.2%
	4	NA	NA	NA	NA
SPINS	2	81.8%	7.5%	10.7%	85.6%
	3	96.7%	2.7%	0.5%	98.9%
	4	99.4%	0.6%	0.0%	100.0%
SPINV	2	70.9%	22.6%	6.5%	95.0%
	3	85.5%	12.3%	2.2%	97.2%
	4	NA	NA	NA	NA
平均	2	74.8%	14.5%	9.05%	92.0%
	3	86.8%	11.3%	1.9%	97.1%
	4	94.6%	5.1%	0.4%	100.0%

ルの割合を示す．表中の「＞ BEN 」は BEN より上昇したことを，「= BEN 」は BEN と同じであることを,「＜ BEN 」は BEN より減少したことを表している．また「= BEN = 1」の値は，FROGa と BEN の *AveP* が同じである場合のうち，その数値がどちらも 1 であった割合を表している．ここから，ほぼ全ての場合に FROGa を使用することで精度が向上するか，あるいは維持されることが観察できる．また精度が維持される場合，両手法で $AveP = 1$ である場合がほぼ全てを占めることが分かる．これは FROGa に効果がないのではなく，BEN がすでに完全な精度での順位付けに成功していたため FROGa による改善の余地がなかったことを意味している．一方で，BEN と比較して精度が落ちた場合はほとんど存在しなかった．

図 3.4.1 のグラフは，SUT ごとの top-*k*% accuracy を示している．グラフの横軸は k の値を，横軸は top-*k*% accuracy の値を表している．このグラフがより早く 1 に達するほど，ランキングの上位から順に候補スキーマを確認していった場合に全ての不具合誘発スキーマが早く見つかることを意味する．候補スキーマの母数が異なるため，異なる *t*-way テスト間でグラフを単純に比較することが出来ないことに注意する必要がある．結果として，FROGa がどの SUT でも BEN より早く top-k% accuracy が 1 に達することが分かる．また FROGa 同士での比較では，対象とする *t*-way テストの網羅次数が高いほど収束が早くなった．これは MAP と同様の傾向を示している．

以上の結果から，RQ1「FROGa は BEN と比較して不具合誘発スキーマとしての疑わしさを順位付けする精度を向上させるか？」に対し，次の回答が得られる．

RQ1 への回答

FROGa は，BEN と比較して候補スキーマの疑わしさを順位付けする精度を大幅に向上させることができる．特に不具合誘発スキーマが 1 つのみ存在する場合，FROGa はほぼ確実にそのスキーマをランキングの頂点に位置付けることができる．また，入力値の多いシステムおよび網羅度の高い組み合わせテストを対象とした不具合誘発入力値組限局では BEN の精度は低くなるが，FROGa の精度は高いまま保たれる．

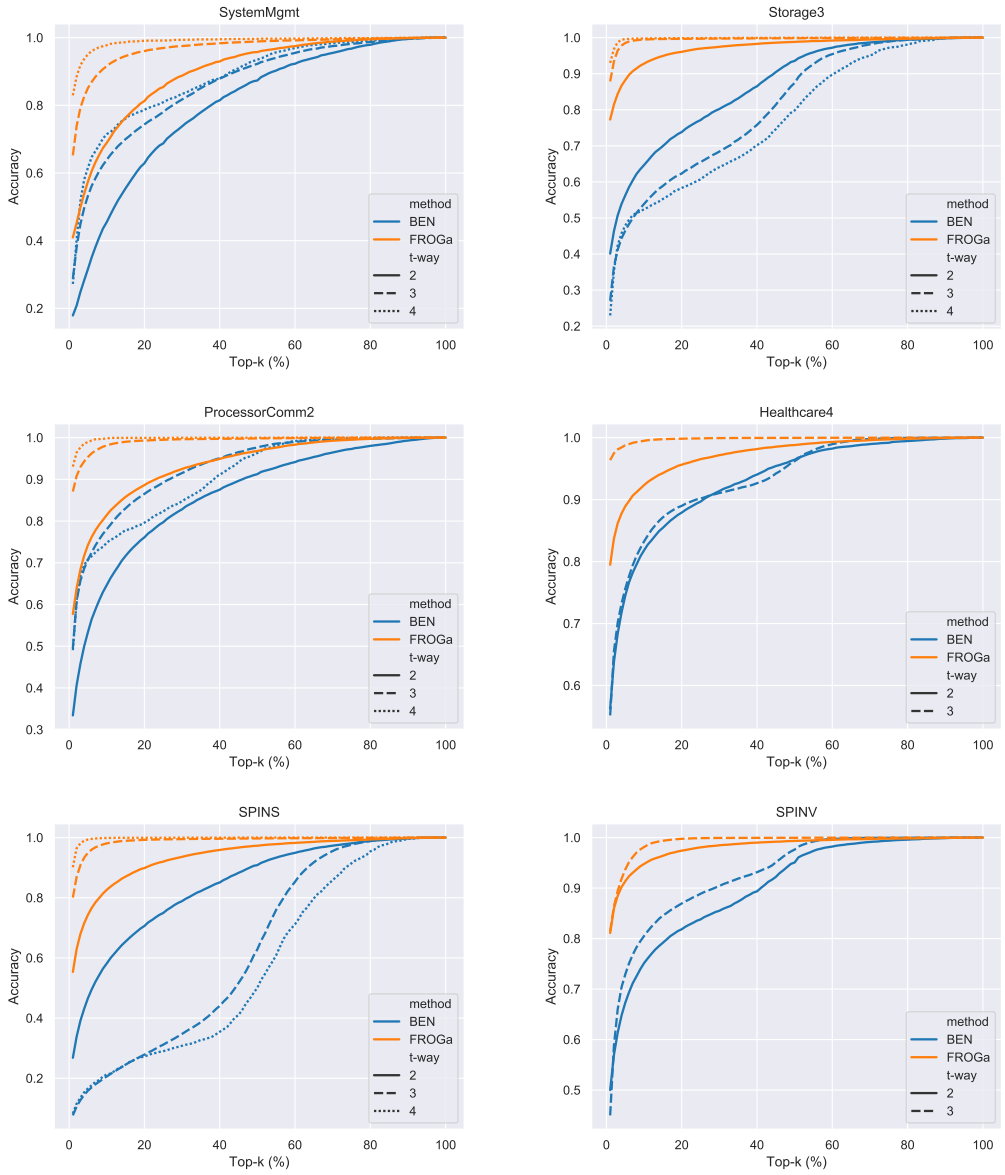

図 3.2: 各 SUT の Top-k% accuracy グラフ

表 3.10: BEN および FROGa における平均処理時間（秒）の比較

	2-way		3-way		4-way	
SUT	BEN	FROGa	BEN	FROGa	BEN	FROGa
SystemMgmt	0.006	0.006	0.035	0.046	0.201	0.373
Storage3	0.019	0.019	0.227	0.348	3.188	6.183
ProcessorComm2	0.028	0.020	0.436	0.531	13.610	19.560
Healthcare4	0.071	0.047	2.423	2.721	NA	NA
SPINS	0.015	0.012	0.188	0.218	3.033	4.589
SPINV	0.148	0.089	8.815	8.688	NA	NA
平均	0.048	0.032	2.021	2.092	5.008	7.769

3.4.2 RQ2 に関する実験結果

表 3.10 に BEN と FROGa における処理時間の平均値の比較を示す．また，図 3.4.2 に両手法における処理時間の分布の比較をボックスプロットにより示す．上側のグラフは各 SUT ごとに比較し，下側のグラフは各 *t*-way テストごとに比較している．図 3.4.1 の場合と同様に，これらには欠損値である Healthcare4 と SPINV の 4-way テストを対象とした場合の結果は含まれない．結果として，BEN と FROGa の間には処理時間の差がほぼ存在しないことが確認できる．FROGa が BEN と比較して性能に優れる一方で時間コストに差がないことから，総合的に FROGa は BEN よりも優れた手法であると結論づけられる．

以上の結果から，RQ2「FROGa と BEN の時間コストに差はあるか？」に対し，次の回答が得られる．

RQ2 への回答

BEN と FROGa の時間コストの間には，明らかな違いはない．

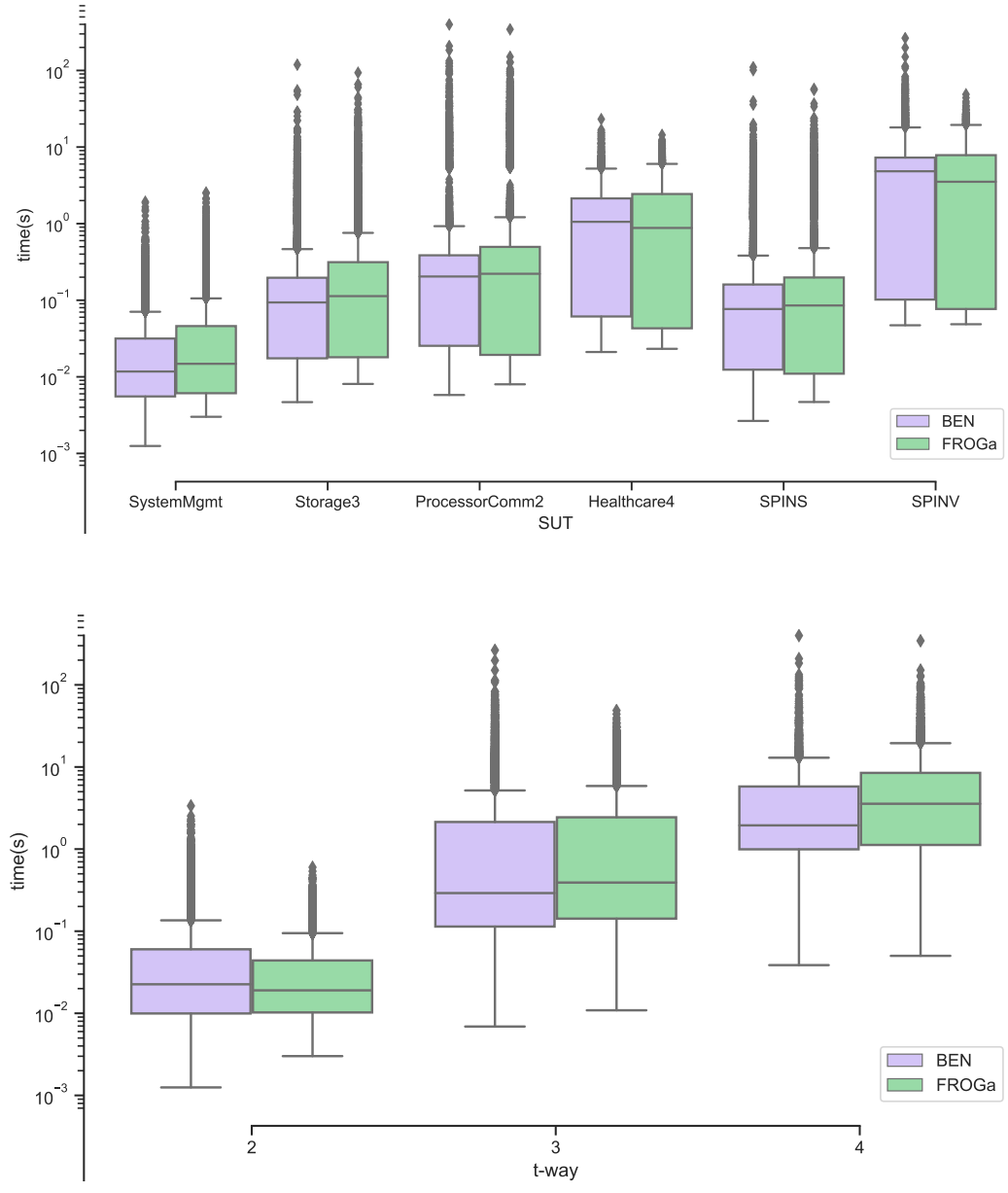

図 3.3: BEN および FROGa における処理時間分布の比較

第 4 章

入力値限局における部分集合の推定による計算効率の改善

4.1 提案モデル：FROGb

4.1.1 概念構成

入力値限局において BEN や FROGa など，全ての候補スキーマを不具合誘発スキーマの候補として扱うアプローチはシンプルかつ徹底的なものである．しかし，そのテストスイートに含まれる全ての候補スキーマを抽出する必要があるため，考慮すべき入力パラメータ数およびその組み合わせ数が増加すると組み合わせ爆発を起こす場合があり，膨大な計算コストが必要になることがある．例えば前章の評価実験では，比較的大きな組み合わせテストである Healthcare4 と SPINV の 4-way テストに両手法を適用した場合，どちらも少なくとも 1 時間以内には候補スキーマの抽

出が終了しなかった．このように現実的でない時間コストは，効率的な欠陥修復を目的として入力値限局を行う意義を失わせてしまう．

組み合わせ爆発を回避するには，全ての候補スキーマを扱うことなく，特に疑わしい候補スキーマのみを直接抽出できることが期待される．この目的のために，ロジスティック回帰分析を再び利用できると考える．

当然ながら，不具合誘発スキーマのサブスキーマを含むテストケースはそうでないテストケースに比べて不具合誘発スキーマを含む可能性が高い．また不具合誘発スキーマを含むテストケースは失敗するため，不具合誘発スキーマのサブスキーマを含むテストケースはそうでないテストケースと比べて失敗する可能性が高い．したがってあるスキーマ s_x の回帰係数は s_x がテストケースに含まれていることがテスト失敗に及ぼす影響の度合いを表すため，不具合誘発スキーマのサブスキーマの回帰係数は高くなると予想される．逆にあるスキーマ s_x の回帰係数が高い場合，s_x は不具合誘発スキーマおよびそのサブスキーマであることが期待される．s_x が不具合誘発スキーマであるためには候補スキーマである必要があるため，s_x が候補スキーマではないにも関わらず高い回帰係数を持っている場合，s_x はより高い次数の不具合誘発スキーマのサブスキーマであると予測できる．

ここから，我々は次の仮説を立てる．

- 不具合誘発スキーマの全てのサブスキーマは 0 より高いロジスティック回帰係数を持つ．

この場合のロジスティック回帰係数は，FROGa と同様にテストケースとスキーマの包含関係およびテスト結果からロジスティック回帰を行うことで得られる回帰係数を指している．回帰係数の閾値を 0 としたのは，回帰係数が正であることはその独立変数の単位増加が従属変数の正値確率に正の影響を与えることを踏まえて，最も感度を広く取ることを目的としている．この仮説が正しければ，サブスキーマと推定されるスキーマが十分に限られる場合，限られたサブスキーマを全て含むようなスーパースキーマは非常に限られるものになる．このことを用いて，サブスキーマとして最初に取得した 1 次スキーマから包含関係を満たすスーパースキーマを次々に取得することによって，探索空間を大きく削減しながら不具合誘発スキーマに効率的に到達できることが期待される．我々はこの考えに基づき，全ての候補スキーマ

を抽出することなく，不具合誘発スキーマとして非常に疑わしい少数の候補スキーマを取得するアルゴリズムである FROGb を提案する．

4.1.2 モデル仕様

FROGb への入力は FROGa と同様に，テストスイートとそれぞれのテストケースのテスト結果，および想定する最大のスキーマ次数 k を必要とする．テスト結果は，何らかのテストオラクルによって成功および失敗に分類されていることを前提とする．またスキーマ次数 k は基本的に，入力するテストスイートが網羅する組み合わせ数を指定する．

以下にアルゴリズムの動作内容を説明する．このアルゴリズムは $1 \leq t \leq k$ となる整数の変数 t に対して，以下のステップを順に実行する．

初期状態

S は現在焦点を当てているスキーマの集合，P は特に不審な候補スキーマの集合，$SubS$ は特に不審な候補スキーマのサブスキーマだと思われるスキーマの集合を意味する．開始時点ではこれらは全て空である．また t の初期値は 1 である．

ステップ 1（$t = 1$）

失敗したテストケースに含まれる 1 次スキーマを全て抽出し S に追加する．

ステップ 2

S の各スキーマについて，それが候補スキーマであるかを定義に従い確認する．候補スキーマであれば S から削除して P に追加する．

ステップ 3

S に残された各スキーマについて，ロジスティック回帰分析によって得られる回帰係数を算出し，回帰係数が正の値を取るスキーマを $SubS$ に追加する．この回帰係数は FROGa での手順と同様に，各スキーマとテストケースとの包含関係およびテストの成否をエンコードし，それを入力としてロジスティック回帰を行うことで得られる．

ステップ 4

t を 1 増加させ，次に焦点を当てるスキーマの次数を上げる．また S を初期化して空にする．その後，ステップ 1 に移る．

ステップ 1（$t \geq 2$）

直前で求めた $t-1$ 次スキーマの集合である $SubS$ に，その全ての $t-1$ 次サブスキーマが含まれるような t 次スキーマを求めて S に追加する．その後，$SubS$ を初期化して空にし，ステップ 2 に移る．

本アルゴリズムの終了条件は次の 2 つである．

1. 第一に，変数 t が所与の最大スキーマ次数 k を超えたとき，すなわち $t > k$ となった場合に終了する．ただし，次数 $k+1$ の候補スキーマを探索する必要はないため，次数 k のスキーマが不審な候補スキーマのサブスキーマか否かを調べる必要はない．$t = k$ の場合に t 次スキーマである S 内の各スキーマが候補スキーマであるかを判定するステップ 2 が終了したのち直ちに終了する．

2. 第二に，$t \leq k$ であるにも関わらず，空の S や $SubS$ に新たな t 次スキーマが追加されず，次のステップに渡すスキーマが存在しない場合に終了する．例えば，$t-1$ 次の全てのサブスキーマが $SubS$ に含まれるような t 次スキーマが存在しない場合，S に t 次スキーマが追加されず直ちに終了する．また S 内の全てのスキーマが候補スキーマであった場合にも，それがより次数の高い不審な候補スキーマのサブスキーマであるか推定を行うスキーマが存在しないため，直ちに終了する．

アルゴリズムが終了したときの P を，k 次スキーマ以下の非常に疑わしい候補スキーマの集合として出力する．

4.1.3 適用例

本モデルの理解を深めるため，表 2.2 に示すテスト結果の例を対象とした FROGb の適用例を示す．表 4.1 に，この適用例における各反復で扱われるスキーマおよびその各種判定結果を示す．この表の $s \in S$ は S に含まれるスキーマを，to P は候補ス

キーマであるか否かを，to *SubS* はサブスキーマとして推定されたか否かを表している．入力として，表 2.2 に示すテストスイートとそのテスト結果，および最大スキーマ次数 $k = 3$ を与える．

$t = 1$

まず，失敗したテストケースに含まれる 1 次スキーマを全て列挙し，S に追加する．この例では，(-, -, -, 3, -) を除く全ての 1 次スキーマが S に追加された．次に，S 内の全ての 1 次スキーマは合格したテストケースにも含まれているため，候補スキーマでないことを確認した．また次に S 内の各スキーマと各テストケースとの包含関係をエンコードし，ロジスティック回帰を行って回帰係数を得る．ここで，5 つの 1 次スキーマ (1, -, -, -, -)，(-, 2, -, -, -)，(-, -, 1, -, -)，(-, -, -, 2, -)，(-, -, -, -, 1) は回帰係数が 0 より高くなったため，不具合誘発スキーマの 1 次サブスキーマ候補として *SubS* に追加された．これらは実際に不具合誘発スキーマのサブスキーマであることが確認できる．ここで S を空にする．

$t = 2$

次に，失敗したテストケースに含まれつつ，その全ての 1 次サブスキーマが *SubS* に含まれている全ての 2 次スキーマを列挙し，空の S に追加する．その結果，表 4.1 に示す 9 つの 2 次スキーマが S に追加された．その一例として，2 次スキーマ (1,2, -, -, -) は表 2.2 の失敗したテストケース#8 である (1,2, -, -, -) に含まれており，その全ての 1 次サブスキーマ (1, -, -, -, -) と (-, 2, -, -, -) が *SubS* に含まれているため，S に追加された．ここで *SubS* を空にする．次に，各 2 次スキーマが候補スキーマであるかを確認する．その結果，(1, -, -, 2, -) が候補スキーマであることが分かったため，S から取り除き，P に追加した．(1, -, -, 2, -) が S から取り除かれたことで，今後 (1, -, -, 2, -) をサブスキーマに含むスキーマが候補スキーマになることはなく，これは不具合誘発スキーマが最小次数であることを満たす．S の残りについて回帰係数を求め，6 つの 2 次スキーマが *SubS* に追加された．ここで S を空にする．

$t = 3$

$t = 2$ の場合と同様に，失敗したテストケースに含まれつつ，全ての 2 次サブ

表 4.1: 各反復で扱われるスキーマおよびその各種判定結果

反復	$s \in S$	to P	回帰係数	to $SubS$
$t = 1$	(1, -, -, -, -)		0.049098	✓
	(2, -, -, -, -)		-0.541751	
	(-, 1, -, -, -)		-0.534167	
	(-, 2, -, -, -)		0.041515	✓
	(-, -, 1, -, -)		0.015767	✓
	(-, -, 2, -, -)		-0.508419	
	(-, -, -, 1, -)		-0.248424	
	(-, -, -, 2, -)		0.570793	✓
	(-, -, -, -, 1)		0.449722	✓
	(-, -, -, -, 2)		-0.447605	
	(-, -, -, -, 3)		-0.494769	
$t = 2$	(1, 2, -, -, -)		0.031674	✓
	(1, -, 1, -, -)		-0.107129	
	(1, -, -, 2, -)	✓	-	
	(1, -, -, -, 1)		-0.106414	
	(-, 2, 1, -, -)		0.585448	✓
	(-, 2, -, 2, -)		0.095438	✓
	(-, 2, -, -, 1)		0.571152	✓
	(-, -, 1, 2, -)		0.095438	✓
	(-, -, 3, -, 1)		0.594410	✓
$t = 3$	(-, 2, 1, 2, -)		-	
	(-, 2, 1, -, 1)	✓	-	

スキーマが $SubS$ に含まれている全ての 3 次スキーマを列挙し，空の S に追加する．そのようなスキーマは (-, 2, 1, 2, -) と (-, 2, 1, -, 1) の 2 つしかないことが分かる．これらが候補スキーマであるかを確認すると，(-, 2, 1, -, 1) のみが候補スキーマであったため，S から削除し P に追加する．ここで，最大次数 $k = 3$ であるため，第一の終了条件が満たされ，アルゴリズムが終了する．

アルゴリズム終了後，その全てのサブスキーマが正の回帰係数を持つ 2 つの非常に不審な候補スキーマのみが P として得られた．これらの候補スキーマは確かに不具合誘発スキーマと一致していることが確認できる．

適用例で示された結果及び計算プロセスから，本手法が候補スキーマをベースとした入力値限局手法の効率化に 2 つの貢献を与えることが読み取れる．第一に，考慮すべき候補スキーマの削減である．今回の例では，BEN と FROGa は全ての 3 次以下の候補スキーマに対して疑惑値を求める必要があったが，FROGb では候補スキーマを非常に疑わしい 2 つのスキーマに限定できた．このことは，膨大な候補スキーマをランク付けすることによる計算コストを回避できる．第二に，あるスキーマが候補スキーマであるかを確認する操作回数の大幅な削減である．BEN と FROGa では，失敗したテストに含まれる 35 個の 3 次スキーマ全てに対してそれが候補スキーマであるかを確認する必要があった．対して FROGb では，サブスキーマの制約を満たすスキーマのみを対象とするため，候補スキーマであるかを確認する 3 次スキーマを 2 つに絞ることができた．これにより，スキーマ次数の増加に伴って爆発的に増加する組み合わせの全てを扱わずに済むため，計算コストを大幅に削減できる．また今回例として使用した SUT モデルが非常に小規模であり，組み合わせテストの網羅度も高くはないことを踏まえると，より大規模な組み合わせテストを対象とした場合にはより高い効率削減効果が期待できる．

4.2 評価方法

実験によって提案手法 FROGb の有効性を評価する．このために以下の研究設問を設定した．

- RQ3：FROGb は，BEN および FROGa に比べて非常に不審な候補スキーマを

抽出するコストをどれだけ削減できるか？

- RQ4：FROGb は，非常に不審な候補スキーマとして不具合誘発スキーマをどれだけ正確に抽出できるか？

FROGb の抽出結果を BEN 及び FROGa と比較するために，前章の結果を再利用し比較する．したがって，実験環境及び実験対象は 3.3 節に示したものと同様である．我々は新たに FROGb を実装し，前章と同様の対象に FROGb を実行することで得られた結果について，次に述べる評価指標をすでに得られた BEN 及び FROGa の結果と比較する．

RQ3 に答えるため，FROGb が終了するまでの処理時間を測定した．FROGb の目的である「非常に疑わしい候補スキーマを抽出する」という操作は，BEN および FROGa を使用した場合には「全てのスキーマから候補スキーマを抽出し，それらの疑惑値を計算してランキングを作成し，ランキング上位から任意の数の候補スキーマを取り出す」という操作によって達成される．したがって，FROGb に入力を与えてから出力が得られるまでの処理時間は，BEN および FROGa におけるランキング作成完了までの処理時間と適切に比較される．またそれぞれの手法が扱うスキーマの数を比較するため，FROGb が抽出した非常に不審な候補スキーマの個数と，あるスキーマが候補スキーマであるかを確認する操作回数を集計した．

次に RQ4 に答えるため，FROGb の出力結果それぞれについて，事前に設定された不具合誘発スキーマが非常に不審な候補スキーマの集合に含まれているか否かを調査した．

4.3 評価結果

4.3.1 RQ3 に関する実験結果

表 4.2 に BEN，FROGa，FROGb の処理時間の平均値および比較のための 2 つの指標を示す．指標 *%red* は，BEN/FROGa と比較したときの FROGb の時間減少率を表す．また指標 *#short* は，FROGb が BEN および FROGa と比較して処理時間を短縮できたケースの数を表す．これは BEN と FROGa のうち，より短いものと比較した

表 4.2: FROGb および比較手法における処理時間の平均値の比較結果

		処理時間 (秒)			*%red*	*#short*
SUT	*t*-way	BEN	FROGa	FROGb		
SystemMgmt	2	0.006	0.006	**0.004**	11.87%	7,150 / 10,000
	3	0.035	0.046	**0.017**	34.55%	8,335 / 10,000
	4	0.201	0.373	**0.046**	62.68%	962 / 1,000
Storage3	2	0.019	0.019	**0.014**	15.23%	7,491 / 10,000
	3	0.227	0.348	**0.102**	47.91%	9,831 / 10,000
	4	3.188	6.183	**0.334**	83.99%	1,000 / 1,000
ProcessorComm2	2	0.028	0.020	**0.016**	18.10%	8,402 / 10,000
	3	0.436	0.531	**0.143**	63.90%	9,986 / 10,000
	4	13.610	19.560	**0.558**	94.06%	1,000 / 1,000
Healthcare4	2	0.071	0.047	**0.046**	-0.27%	5,139 / 10,000
	3	2.423	2.721	**0.853**	65.21%	9,999 / 10,000
	4	NA	NA	**3.747**	NA	NA
SPINS	2	0.015	0.012	**0.010**	2.89%	6.046 / 10,000
	3	0.188	0.218	**0.082**	47.78%	9,531 / 10,000
	4	3.033	4.589	**0.236**	89.73%	1,000 / 1,000
SPINV	2	0.148	0.089	**0.096**	-14.65%	2,922 / 10,000
	3	8.815	8.688	**2.514**	70.14%	9,939 / 10,000
	4	NA	NA	**31.372**	NA	NA
合計	2				5.53%	37,150 / 60,000
	3				54.92%	57,621 / 60,000
	4				82.62%	3,962 / 4,000

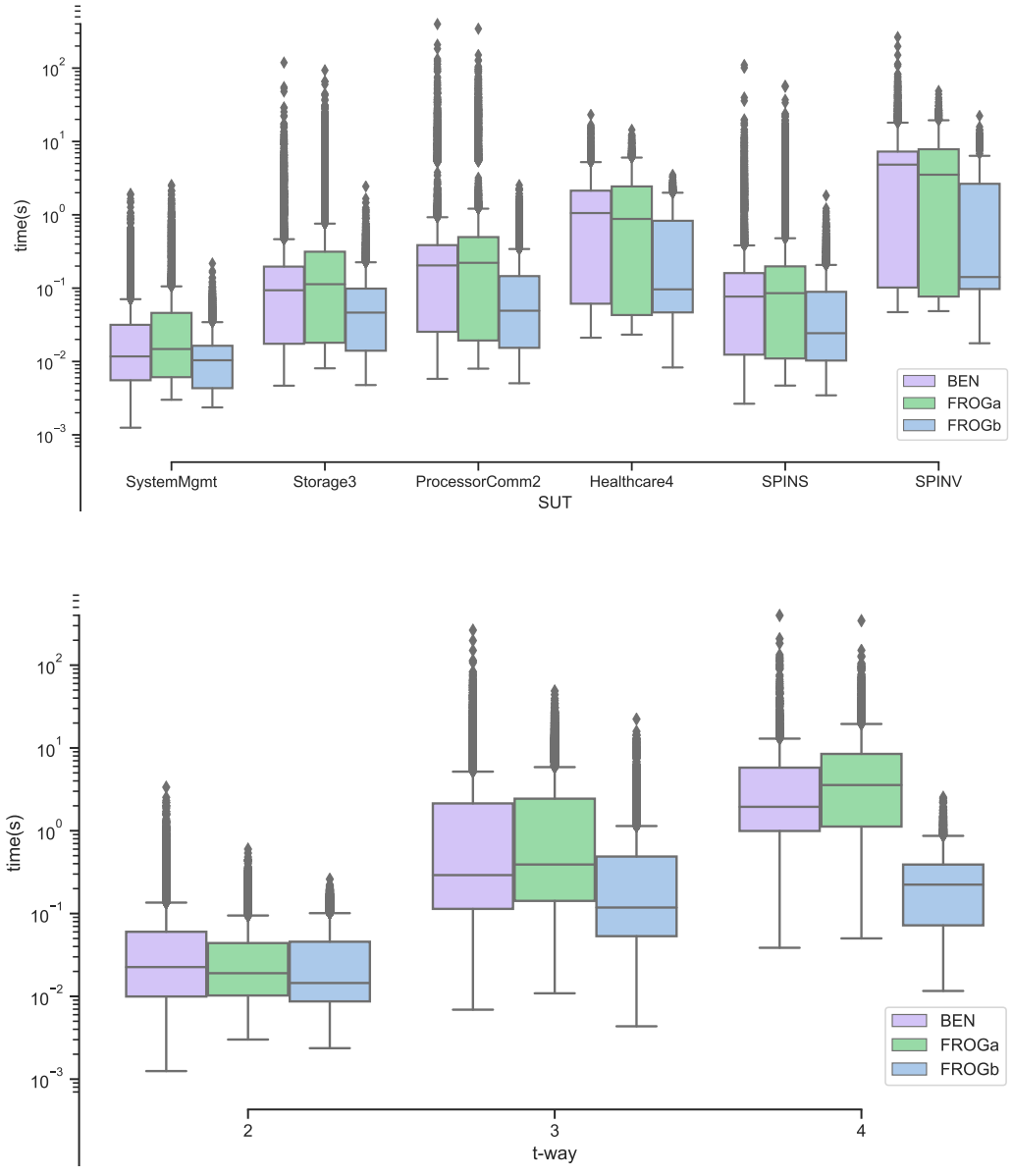

図 4.1: FROGb と比較手法の処理時間の分布比較

結果を用いている．また，図 4.2 に処理時間分布を比較したボックスプロットを示す．この結果として，テストスイートの網羅スキーマ次数による処理時間削減効果の差が顕著に見られた．まず，2-way テストを対象とした場合は FROGb による時間短縮はほとんど見られなかった．特に Healthcare4 と SPINV の場合，平均時間減少率は 0 未満となり，処理時間は半数以上のケースで増加した．この理由として，2 次スキーマの総数がそれほど多くはないため，FROGb による処理時間のオーバーヘッドが FROGb による削減時間を上回ったことが考えられる．次に，3-way テストでは FROGb による時間削減率は約 50%となり，60,000 件の試行のうちほぼ全てとなる 57,621 件で処理時間が短縮された．さらに，4-way テストでは処理時間の短縮がより顕著となった．特に Healthcare4 と SPINV の 4-way テストを対象とした場合，FROGb は数秒で処理を終了できた．BEN と FROGa が 1 時間以内に処理を終了できなかったことを考えると，これは注目すべき改善であると言える．結果を要約すると，対象のテストスイートが網羅するスキーマの次数が高いほど FROGb の処理時間削減効果は高くなった．

以下，簡単のため，FROGb とは異なり先に全ての候補スキーマを抽出する手法である BEN と FROGa をまとめて全抽出法と呼称する．表 4.3 の左側に全抽出法が抽出した候補スキーマ数，および，FROGb が抽出した非常に不審な候補スキーマ数の平均値とその比較を示す．また表 4.3 の右側に，全抽出法および FROGb であるスキーマが候補スキーマであるかを確認した操作回数の平均値とその比較を示す．%*red* は全抽出法と比較した場合の FROGb の減少率を，#*red* は全抽出法と比較して FROGb で減少した試行の数を表している．さらに，図 4.2 の上部には候補スキーマ数の分布比較を，下部には確認操作回数の分布比較を図示する．結果として，FROGb が出力する非常に不審な候補スキーマの数は全抽出法が扱う候補スキーマ数に比べてほぼ全ての場合で減少した．各 t-way テストで分類した場合には，出力される候補スキーマ数の削減率は 50.58%($t = 2$)，91.52%($t = 3$)，98.97%($t = 4$) となり，網羅スキーマ次数が上がるほど削減された．一方，SUT による違いは見られなかった．また FROGb が候補スキーマであるか確認した操作回数は，網羅スキーマ次数が大きくなるにつれて減少することが分かった．これは全抽出法とは真逆の傾向である．この理由は，FROGb アルゴリズムの最大反復数 k が高くなることで，出

表 4.3: FROGb および比較手法が扱う候補スキーマ数の平均値の比較結果

		候補スキーマ数		%red	#red	確認操作回数		%red
SUT	*t*-way	全抽出法	FROGb			全抽出法	FROGb	
SystemMgmt	2	52.4	**26.0**	50.36%	9,962	178.2	**73.5**	58.98%
	3	204.8	**12.5**	92.01%	9,997	1,086.7	**76.4**	92.69%
	4	634.3	**4.7**	98.25%	1,000	3,636.5	**54.8**	98.12%
Storage3	2	93.5	**43.2**	51.63%	9,848	650.0	**224.4**	65.94%
	3	433.1	**21.6**	91.42%	9,978	7,116.5	**229.9**	96.67%
	4	2,305.7	**5.1**	99.03%	1,000	47,122.2	**146.1**	99.61%
ProcessorComm2	2	147.6	**64.1**	55.56%	9,968	1,109.5	**346.5**	68.60%
	3	765.5	**45.6**	92.00%	10,000	21,231.4	**513.8**	97.37%
	4	5,494.4	**5.7**	99.32%	1,000	280,509.3	**258.2**	99.86%
Healthcare4	2	211.9	**98.7**	50.58%	9,909	3,674.7	**1,003.3**	72.57%
	3	1,401.1	**80.5**	92.60%	10,000	127,433.2	**1,828.0**	98.55%
	4	NA	**5.7**	NA	NA	NA	**764.0**	NA
SPINS	2	106.4	**54.2**	48.01%	9,942	706.8	**231.8**	67.64%
	3	482.1	**29.0**	90.32%	9,978	9,082.6	**301.1**	96.60%
	4	2,506.0	**4.9**	99.26%	1,000	78,524.0	**168.0**	99.77%
SPINV	2	268.2	**134.3**	48.98%	9,859	7,438.3	**1,890.5**	74.87%
	3	2,227.0	**135.6**	90.79%	9,985	321,912.1	**5,102.2**	98.42%
	4	NA	**26.3**	NA	NA	NA	**3,032.1**	NA
平均	2			50.85%	59,488			68.10%
	3			91.52%	59,938			96.72%
	4			98.97%	4,000			99.34%

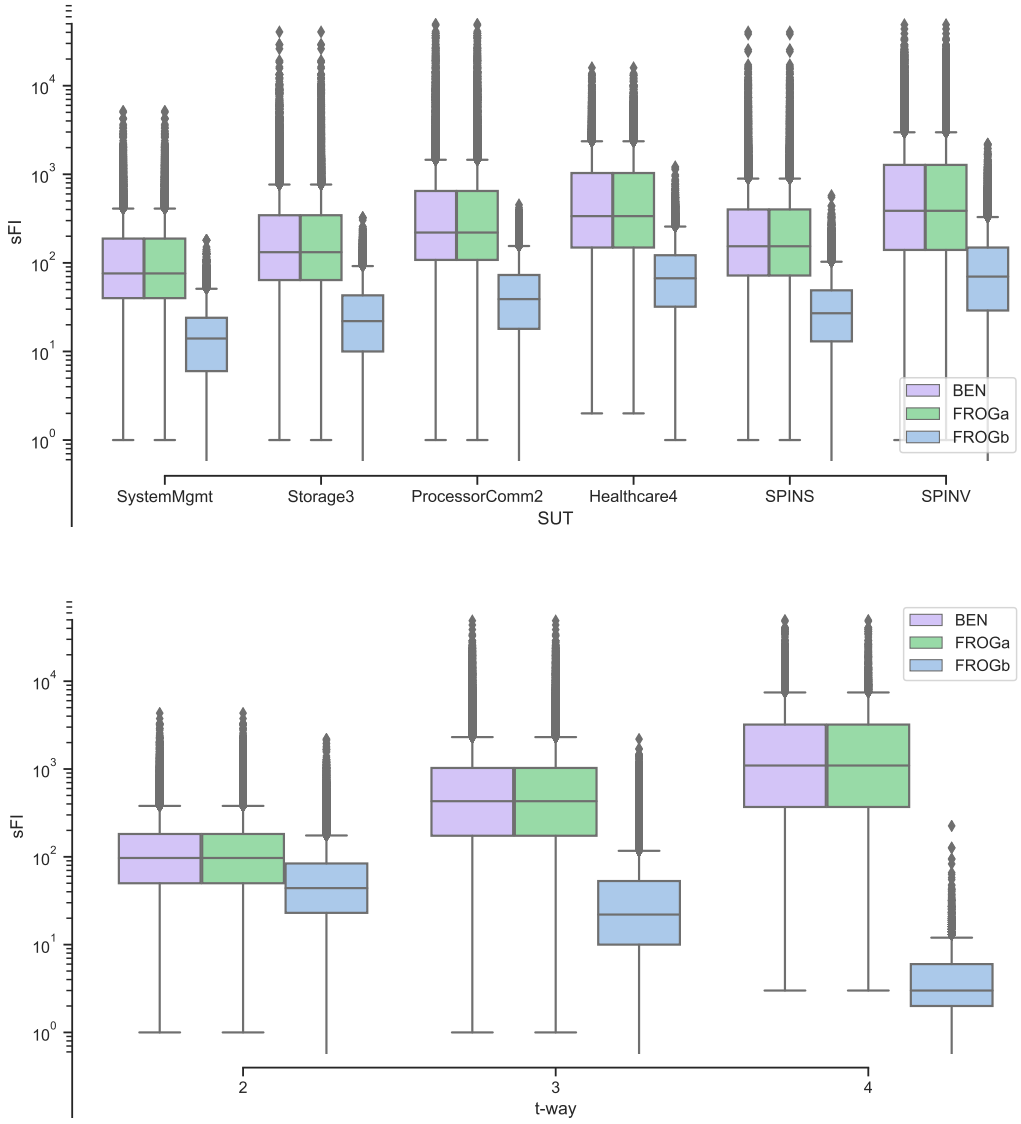

図 4.2: FROGb と比較手法が出力する候補スキーマ数の分布比較（上），および確認操作回数の分布比較（下）

表 4.4: FROGb の出力結果に不具合誘発スキーマが含まれる内訳

	2-way			3-way			4-way		
SUT	All	Partly	No	All	Partly	No	All	Partly	No
SystemMgmt	3,748	5,474	778	3,110	5,276	1,614	396	401	203
Storage3	6,353	3,479	168	5,614	3,949	437	464	439	97
ProcessorComm2	6,228	3,529	243	6,848	2,957	195	626	349	25
Healthcare4	8,227	1,745	28	8,557	1,414	29	781	212	7
SPINS	6,183	3,692	125	6,316	3,507	177	533	411	56
SPINV	8,465	1,526	9	8,386	1,589	25	797	182	21
合計	39,204	19,445	1,351	38,831	18,692	2,477	3,597	1,994	409
(%)	(65.34%)	(32.41%)	(2.25%)	(64.72%)	(31.15%)	(4.13%)	(59.95%)	(33.23%)	(6.82%)

力される候補スキーマが満たす必要のあるサブスキーマの制約が厳しくなるためだと考えられる．ほとんどの SUT で，$k = 4$ に設定した場合に出力される候補スキーマの数の平均値は 5 個程度である．これは，実際に FROGb が最小限の操作回数によって非常に不審な候補スキーマのみを出力する能力を有することを証明している．

以上の結果から，RQ3「FROGb は，BEN および FROGa に比べて非常に不審な候補スキーマを抽出するコストをどれだけ削減できるか？」に対し，次の回答が得られる．

RQ3 への回答

FROGb は，特に網羅次数の高い組み合わせテストを対象とした場合に，BEN や FROGa に比べて処理時間を大幅に短縮しながら，非常に疑わしい少数の候補スキーマのみを抽出することが可能である．

4.3.2 RQ4 に関する実験結果

表 4.4 に，FROGb が出力した少数の非常に不審な候補スキーマ集合に不具合誘発スキーマが含まれるかを調査した結果を示す．結果は，全て含む（All），部分的に含む（Partly），含まない（No）の 3 カテゴリに分類され集計された．表 4.4 の結果から，全ての不具合誘発スキーマが出力結果に常に含まれるわけではないことが分かる．平均すると，全ての不具合誘発スキーマが含まれる割合は全体の 63.3%だっ

表 4.5: n 個の不具合スキーマが存在する場合に FROGb の出力結果に全ての不具合誘発スキーマが含まれていた割合

	2-way			3-way			4-way		
SUT	$n=1$	$n=2$	$n=3$	$n=1$	$n=2$	$n=3$	$n=1$	$n=2$	$n=3$
SystemMgmt	88.56%	46.47%	26.75%	71.32%	32.44%	13.58%	77.81%	23.28%	9.28%
Storage3	96.57%	70.39%	49.01%	92.37%	55.91%	30.36%	88.64%	32.09%	12.89%
ProcessorComm2	94.32%	66.94%	42.53%	96.69%	70.38%	39.91%	98.34%	58.89%	23.05%
Healthcare4	99.48%	86.90%	65.32%	99.54%	90.76%	66.65%	99.44%	80.90%	50.80%
SPINS	97.67%	71.08%	45.07%	99.39%	67.15%	34.79%	92.95%	49.23%	22.87%
SPINV	99.93%	88.09%	68.70%	99.78%	90.27%	61.73%	94.69%	84.15%	60.06%
平均	96.09%	71.64%	49.56%	93.18%	67.82%	41.17%	91.98%	54.76%	29.83%

た．また 32.2%の割合で一部の不具合誘発スキーマのみが含まれ，どの不具合誘発スキーマも含まれない割合は 4.4%のみであった．この内訳について，対象とするテストスイートの網羅スキーマ次数による違いはほとんどないことが分かる．一方，対象とする SUT の入力規模が大きいほど全ての不具合誘発スキーマを非常に不審な候補スキーマとして抽出可能な割合が高い傾向が見られた．

表 4.5 は，存在する不具合誘発スキーマの個数ごとに集計された，FROGb の出力結果が全ての不具合誘発スキーマを含む割合を示している．ここから，不具合誘発スキーマ数が少ないほど全ての不具合誘発スキーマを抽出できる割合が上がることが確認できる．また不具合誘発スキーマの数が増えると，成功率は格段に下がっていくことが分かる．例えば SPINV の 2-way テストを対象とした場合，不具合誘発スキーマが 1 個のみ存在する場合には 99.9%と非常に高精度に不具合誘発スキーマを非常に不審な候補スキーマとして抽出できたが，不具合スキーマが 3 個存在する場合は 68.7%にまで急激に精度が下がった．これは，単純に複数の不具合誘発スキーマが全て抽出に成功する確率的な難しさだけによるものではなく，不具合誘発スキーマが複数存在することが抽出失敗に影響を与えることを示唆している．また SUT の持つ入力規模の差による抽出精度の差は，不具合誘発スキーマの個数によってさらに強調された．3 個の不具合誘発スキーマが存在する場合，入力空間の大きい Healthcare4 や SPINV での全抽出成功率は約 50%から約 70%だったが，入力空間の小さい SystemMgmt では約 9%から約 27%と極端に低いものとなった．

以上の結果から，RQ4「FROGb は，非常に不審な候補スキーマとして不具合誘発スキーマをどれだけ正確に抽出できるか？」に対し，次の回答が得られる．

RQ4 への回答

FROGb は必ず全ての不具合誘発スキーマを非常に不審な候補スキーマとして抽出できるわけではないが，全体の 63.3%で不具合誘発スキーマを全て抽出でき，また全体の 95.6%で少なくとも部分的に抽出できる．この精度は，SUT の入力規模が大きいほど，また不具合誘発スキーマの個数が少ないほど高くなる傾向にある．

4.4 議論

4.4.1 総合的な評価

RQ2 への回答から，我々の仮説に反し，不具合誘発スキーマの全てのサブスキーマが常に正の回帰係数を持つわけではないことが明らかになった．したがって，この仮説を基に構築された FROGb は必ずしも正確な結果を保証しない．しかし，実験では約 67%の割合で全ての不具合誘発スキーマを少数の非常に不審な候補スキーマとして抽出できることが分かった．この成功率は決して低いものではなく，むしろ従来の手法では非現実的な処理時間となるような適用対象に対しては，処理時間削減との優れたトレードオフであると言える．また，一部の不具合誘発スキーマを限局できるだけでも価値のある結果となる場合もある．例えば，複数の不具合誘発スキーマが単一の欠陥を誘発している場合，いずれかの不具合誘発スキーマを特定するだけで欠陥を修復できる可能性がある．また，一部の不具合誘発スキーマの特定によって一部の欠陥が修復できた場合，組み合わせテストを再度行うことで異なるテスト結果が得られることによって，連鎖的に全ての不具合誘発スキーマおよびそれらが誘発する欠陥箇所を限局できる可能性もある．そのため，部分的な正確さを許容すれば，FROGb は約 96%の割合で効果的な限局結果を素早く得ることができる方法であると評価できる．以上の考えをもとに，我々は FROGb を非常に効率のよい近似的な入力値限局アプローチであると結論づける．

4.4.2 失敗原因の考察

FROGb が全ての不具合誘発スキーマの抽出に失敗したいくつかの試行の処理内容を目視確認することで，2 つの失敗原因を特定した．

第一の理由は，複数の不具合誘発スキーマの「衝突」によるものである．これは，複数の不具合誘発スキーマが存在し，あるパラメータの取りうる全ての値に対応した全ての 1 次スキーマがそれぞれ別の不具合誘発スキーマに含まれている状態を指す．FROGb はこれらの 1 次スキーマの回帰係数が全て正の値を持つことを期待しているが，回帰係数の相対性から，これらの 1 次スキーマのうち一部のみしか正の値を持つことができないため，全ての 1 次スキーマを疑わしい候補スキーマのサブスキーマとして推測することができず，したがっていずれかの不具合誘発スキーマに到達できない．我々の事後調査では，すべての不具合誘発スキーマを抽出できなかった 44,368 件の試行のうち，12,319 件で「衝突」を伴う複数の不具合誘発スキーマが存在していた．また「衝突」を伴う複数の不具合誘発スキーマを含む試行は全て失敗していた．この失敗理由は，SUT の入力規模が小さく不具合誘発スキーマの個数が多いほど全ての不具合誘発スキーマを抽出できる精度が低くなるという実験結果をうまく説明している．本実験において不具合誘発スキーマの割り当てはランダムに決定されているので，不具合誘発スキーマの個数が増えると「衝突」が起きる可能性は上昇する．また SUT の入力パラメータ数が多く，多様な値をとれる場合，「衝突」の可能性は下がる．

第二の理由は，単なる偶然によるものである．これは，設計されたテストスイートの各テストケースにどの入力値が偶然に割り当てられたかを指す．次に示す 2 通りの偶然による限局失敗事例が確認できた．

1. 複数の不具合誘発スキーマが存在している．ある不具合誘発スキーマを構成するあるパラメータについて，それが別の値に割り当てられているテストケース全てに他の不具合誘発スキーマが偶然含まれている場合，見かけ上，そのパラメータの値は問われないことになる．したがって実際の不具合誘発スキーマから 1 次分少ないスキーマが候補スキーマとして見かけ上成立してしまうため，誤った結果が出力される．

2. 複数の不具合誘発スキーマが存在している．ある不具合誘発スキーマに含まれる 1 次スキーマ s を考える．s がテストケースに含まれる場合と比較して，s がテストケースに含まれない場合に別の不具合誘発スキーマが偶然にテストケースに含まれる頻度が高い場合，s に対応する回帰係数は 0 より低くなる．これによって s は実際には不具合誘発スキーマのサブスキーマであるにも関わらずサブスキーマとして推定されなくなり，誤った結果が出力される．

その他の失敗理由は見つけられなかったため，第一の失敗理由で説明できない 32,049 件の失敗は全て第二の失敗理由によると思われる．第一の失敗理由では不具合誘発スキーマの割り当てパターンに依存するため常に失敗するが，第二の失敗理由は設計されたテストスイートのパターンに依存するため，異なる設計方法で得られたテストスイートでは成功する場合もある．

4.4.3 制限

FROGb が持つ既知の制限を挙げる．1 つのテストケースのみが失敗したテストスイートに対する FROGb の適用は推奨されない．これは，失敗したテストケースに含まれる全ての 1 次スキーマが不具合誘発スキーマのサブスキーマとして予測されるためである．したがって結局，そのテストケースに含まれる全ての候補スキーマが出力されるため，欠陥限局に有益な情報をもたらさない．こうした場合は，OFOT 法 [48] などの単一の失敗テストケースの改変と再実行に基づく入力値限局手法の使用がより推奨される．

また，FROGb を使用した場合には全ての候補スキーマが明らかでないため，他の全ての候補スキーマを排したテストケースを用いて疑わしい候補スキーマが不具合誘発スキーマであることを確定させることはできない．ただし必要に応じて，ICT [50] で採用されているチェッキング機構のように疑わしい候補スキーマを含むランダムなテストケースを何度かテストすることで確信を深めることは可能である．

第 5 章

入力値限局における実行順序に依存したテスト結果への拡張

5.1 周辺知識

5.1.1 OFOT 法

代表的な入力値限局手法として，Nie らは OFOT（One Factor One Time）法を提案している [48]．この手法では単一の失敗したテストケースを入力として使用し，そのパラメータ値を一つずつ変更して再びテストすることで不具合誘発スキーマを特定する．これは，あるパラメータ値を変更することによってテスト結果が失敗から合格に変わる場合，そのパラメータ値が不具合誘発スキーマの一部だと分かることを利用している．

表 5.1: 例題システムの入力パラメータ

パラメータ	取りうる値
p_a	0, 1, 2
p_b	0, 1, 2
p_c	0, 1
p_d	0, 1

表 5.2: 例題システムの 2-way テストスイート例

テストケース	p_a	p_b	p_c	p_d	テスト結果
t_1	0	0	0	0	失敗
t_2	0	1	1	1	合格
t_3	0	2	1	0	合格
t_4	1	0	0	1	合格
t_5	1	1	0	0	失敗
t_6	1	2	1	1	合格
t_7	2	0	1	1	合格
t_8	2	1	0	0	失敗
t_9	2	2	0	0	失敗

本章全体を通して例示に使用する，より簡素な例題システムの SUT モデルを表 5.1 に示す．また PICT により生成されたこの例題システムの 2-way テストスイート，および， (-, -, 0, 0) が不具合誘発スキーマである場合のテスト結果を表 5.2 に示す．

表 5.2 に示した組み合わせテスト結果への OFOT 法の適用例を以下に示す．失敗したテストケース t_1 に注目し，表 5.3 に示すように各パラメータごとに値を変更した新しいテストケース t_{a1} から t_{a4} を作成し実行する．その結果，t_{a1} と t_{a2} は依然としてテストに失敗するが，t_{a3} と t_{a4} はテスト結果が合格に変化した．このことから，テストケースがテストに失敗するためには，(-, -, 0, -）および (-, -, -, 0) が含まれる必要があることが分かる．したがって，(-, -, 0, 0) が不具合誘発スキーマとして特定さ

表 5.3: OFOT 法の適用例で使用される追加テストケース

テストケース	p_a	p_b	p_c	p_d	テスト結果
t_{a1}	1	0	0	0	失敗
t_{a2}	0	1	0	0	失敗
t_{a3}	0	0	1	0	合格
t_{a4}	0	0	0	1	合格

れる．これは実際にシステムが失敗する条件に一致していることが確認できる．さらに，この不具合誘発スキーマの存在によって t_1 以外のすべてのテストの失敗も説明できるため，不具合誘発スキーマは他に存在しないと見なすことができる．

5.1.2 Flaky Test

Micco は非決定的な出力結果をもたらすテストについて報告し，Flaky Test と名付けている [45]．また，Luo らは 51 件の Apache オープンソースプロジェクトから得られた 210 の Flaky Test を 10 種類の要因に分類した [42]．その結果，非同期待機，同時実行，順序依存性が上位 3 つの原因であることを明らかにした．

既存の入力値限局手法では同一のテストケースからは常に同一の結果が得られるといった仮定をおくなど，Flaky Test の存在に対応しておらず，その有効性に限界がある．Niu らの研究 [50] は，彼らの入力値限局手法を評価する際にテスト結果の非決定性を考慮した，我々の知る唯一の研究である．彼らは同じテストケースを複数回実行することで非決定性の緩和を試みたが，確率的な方法がゆえに効果は不確実であり，また完全なランダム性に由来する非決定性にのみ機能する．

我々はテスト結果の非決定性を引き起こす上位の要因のうち，順序依存性に焦点を当てる．これについてアルゴリズムによる解決が可能であることを本論文で示すことで，入力値限局手法の持つ限界の拡張を試みる．

5.2 提案手法：F-CODE

5.2.1 前提

手法の提案にあたり，以下を前提とする．

- テスト対象のシステムは，テストスイートとして構成されている一連のテストケースを順に連続して実行できる．このとき，システムは特定のテストケースを実行したことによる特定の状態の変化を維持したまま次のテストケースを実行する．テストスイートの全てのテストケースの実行が終了するとシステムの状態は初期化される．

- すべての不具合誘発スキーマには，対応するトリガスキーマが一つ存在する．トリガスキーマは，それを含むテストケースを実行すると，対応する不具合誘発スキーマを含む後続のテストケースが失敗するスキーマを指す．対応する不具合誘発スキーマとトリガスキーマのペアを本論文ではスキーマチェーンと呼称する．

- 不具合誘発スキーマを含むテストケースは，対応するトリガスキーマを含む一つ以上のテストケースがそれ以前に実行されている場合に限り必ず失敗する．そうでない場合には必ず成功する．

5.2.2 モデル仕様

F-CODE は実行されたテストスイートとその結果を入力として受け取り，特定されたすべてのスキーマチェーンを出力する．

まず，失敗したテストケースのリスト L_{fail} からテストケースを取り出し，その失敗を引き起こしたスキーマチェーンの特定作業に移る．特定作業は以下の 3 つのフェイズで構成される．ただし，既に発見されたスキーマチェーンがそのテストの失敗を説明できる場合，特定作業をスキップする．

フェイズ 1：トリガスキーマを含むテストケースの特定

失敗したテストケースを t_{fail} とする．このフェーズでは，t_{fail} に含まれる不具

合誘発スキーマに対応するトリガスキーマを含むテストケースを特定する．入力された一連のテストケースで t_{fail} よりも前に実行されるテストケースを順に選び，そのテストケースと t_{fail} のペアを続けて実行する．t_{fail} がテストに初めて失敗した場合，トリガスキーマは選択したテストケースに含まれていることが分かる．このテストケースを t_{trg} とする．

フェイズ２：不具合誘発スキーマの特定

このフェイズでは，次のようにOFOT法を適用することにより，t_{fail} に含まれる不具合誘発スキーマを特定する．OFOT法によっていずれかのパラメータ値が変更された t_{fail} を実行する際に，t_{trg} の実行に続けて実行させる．これにより，t_{fail} のテスト結果の非決定性が排除されるため，OFOT法が適切に機能する．

フェイズ3：トリガスキーマの特定

このフェイズでは，フェイズ2と同様に t_{trg} にOFOT法を適用することにより，トリガスキーマを特定する．具体的には，OFOT法によって変更された t_{trg} に続いて実行される t_{fail} の実行結果の変化によってトリガスキーマを特定できる．

フェイズ1およびフェイズ3では，特定のテストケースによって不具合誘発スキーマが有効化されるかどうかを調べるため，トリガスキーマを含むと思われるテストケースと不具合誘発スキーマを含むと思われるテストケースの2つのみを連続して実行する必要がある．対してフェイズ2では，トリガスキーマが含まれるテストケースに続けて，追加生成された全てのテストケースを続けて実行できる．これは，追加実行するテストケースが不具合誘発スキーマを含むか否かのみによってテストの成否が決定され，他の追加実行するテストケースの存在に依存しないためである．

これらのフェイズを通じて，t_{fail} の失敗を説明するスキーマチェーンが得られる．L_{fail} がまだ空でない場合，F-CODEは L_{fail} から次の t_{fail} を取得し，その t_{fail} に対してフェイズ1から3を再度実行して，新たなスキーマチェーンを特定する．これを L_{fail} が空になるまで繰り返すことで，すべてのテストケースの失敗を説明するために十分な数のスキーマチェーンが得られる．

表 5.4: F-CODE 適用例への入力に用いる組み合わせテスト結果

テストケース	p_a	p_b	p_c	p_d	テスト結果
t_1	0	0	0	0	合格
t_2	0	1	1	1	合格
t_3	0	2	1	0	合格
t_4	1	0	0	1	合格
t_5	1	1	0	0	失敗
t_6	1	2	1	1	合格
t_7	2	0	1	1	合格
t_8	2	1	0	0	失敗
t_9	2	2	0	0	失敗

5.2.3 適用例

表 5.4 に示されたテスト結果に対して F-CODE を適用する．これは，5.1 節で扱った例題システムの 2-way テストスイートであり，不具合誘発スキーマも同じく (-, -, 0, 0) である．ただし，この不具合誘発スキーマは順序依存性を有しており，対応するトリガスキーマである (-, 2, 1, -) を含むテストケースが実行されない限り，テストは失敗しない．トリガスキーマを含むテストケースは t_3 で初めて実行されるため，t_1 は不具合誘発スキーマを含むにも関わらずテストに失敗していない．

F-CODE はまず，失敗したテストケースのリスト $L_{fail} = \{t_5, t_8, t_9\}$ を取得し，最初に t_5 を取り出す．表 5.5 は各フェイズで作成および実行されたテストケースを示している．フェイズ 1 では，t_5 を失敗させるテストケースを特定するために，t_5 より前に実行されたテストケースと t_5 を続けて順に実行する．t_3 と t_5 を続けて実行した場合に初めて t_5 が失敗したので，t_3 には，t_5 の不具合誘発スキーマを誘発するトリガスキーマが含まれていることがわかる．フェイズ 2 では，OFOT 法を適用するため t_5 の一部を変更した追加テストケースとして t_{a1} から t_{a4} を作成し，t_3 に続けてそれらを順に実行する．$t_{a1}/t_{a2}/t_{a3}/t_{a4}$ の各テストケースは全く依存が無く独立に実行される．この例では t_{a1} や，t_{a2} の実行で失敗しているが，そのあとに実行される t_{a3}

表 5.5: F-CODE 適用例における各フェイズで実行されるテストケース及びその結果

フェイズ	テストケース	p_a	p_b	p_c	p_d	テスト結果
フェイズ 1	t_1	0	0	0	0	-
	t_5	1	1	0	0	合格
	t_2	0	1	1	1	-
	t_5	1	1	0	0	合格
	t_3	0	2	1	0	-
	t_5	1	1	0	0	失敗
フェイズ 2	t_3	0	2	1	0	-
	t_{a1}	2	1	0	0	失敗
	t_{a2}	1	2	0	0	失敗
	t_{a3}	1	1	1	0	合格
	t_{a4}	1	1	0	1	合格
フェイズ 3	t_{a5}	1	2	1	0	-
	t_5	1	1	0	0	失敗
	t_{a6}	0	0	1	0	-
	t_5	1	1	0	0	合格
	t_{a7}	0	2	0	0	-
	t_5	1	1	0	0	合格
	t_{a8}	0	2	1	1	-
	t_5	1	1	0	0	失敗

や t_{a4} の実行に影響しない．したがってそれらのテストケースの実行は成功する．t_{a3} および t_{a4} がテストに通過したことから，t_5 に含まれる不具合誘発スキーマは (-, -, 0, 0) であることがわかる．フェイズ 3 では，トリガスキーマが含まれる t_3 に OFOT 法を適用するためその一部を変更した t_{a5} から t_{a8} を作成し，各テストケースと t_5 が連続するよう順に実行する．t_{a6} および t_{a7} がテストに通過したことから，t_3 に含まれるトリガスキーマは (-, 2, 1, -) であることがわかる．

こうして発見された，不具合誘発スキーマが (-, -, 0, 0) かつトリガスキーマが (-, 2, 1, -) というスキーマチェーンは，他の失敗したテストケース t_8 および t_9 の失敗をも説明できる．したがって L_{fail} が空になり，F-CODE は動作を終了する．この適用例を通じて，F-CODE は 8 つの追加テストケースを作成し，19 のテストケースを追加実行した．

5.3 評価方法

5.3.1 実験概要

F-CODE の性能を評価するために 2 つの実験を行った．

実験 1 システムに単一のスキーマチェーンが存在する場合の組み合わせテストの結果に F-CODE を適用し，そのスキーマチェーンを実際に特定できたかを調べることで，F-CODE の精度，および特定に必要な追加テスト実行数を調査する．実験対象には，いくつかの実際のシステムの入力モデルから人工的にランダムベースで生成された多くの組み合わせテスト結果を使用する．さらに，F-CODE の結果を従来の入力値限局メソッドの結果と比較する．

実験 2 F-CODE は単一のスキーマチェーンの存在を前提として提案されているが，複数のスキーマチェーンが独立に存在する場合に対してもある程度の効果があることが期待できる．このことを確認するため， 特定すべきスキーマチェーンが単一ではなく複数存在する場合について，実験 1 と同様に評価する．

表 5.6: 評価に使用するテスト対象システム

システム名	入力パラメータモデル	パラメータ数
Tomcat	$2^8 \times 3^1 \times 4^1$	10
Hsqldb	$2^9 \times 3^2 \times 4^1$	12
Gcc	$2^9 \times 6^1$	10
Jflex	$2^{10} \times 3^2 \times 4^1$	13
Tcas	$2^7 \times 3^2 \times 4^1 \times 10^2$	12

表 5.7: 各 t-way テストスイートを構成するテストケース数

システム名	$t = 2$	$t = 3$	$t = 4$
Tomcat	14	36	89
Hsqldb	14	43	122
Gcc	14	48	105
Jflex	12	46	120
Tcas	100	404	1,382

5.3.2 実験対象

これまでの先行研究 [27,50,68] では，実際に不具合報告のあったシステムを対象に入力値限局手法を適用することで評価を行っている．しかし本手法が評価したい，順序に依存する不具合の発生とそれを検出したテストケースの双方を備えた不具合報告は我々が調査した範囲では確認できず，したがって実際に行われたテストの再現が不可能である．また先行研究で行われているようなわずか数件の不具合報告を基にした評価では，結果を十分に一般化できないと考える．こうした理由を踏まえ，実際のシステムにおける架空の不具合を多数想定し，それを元に得られたテスト結果に提案手法を適用することで詳細な評価を行う．この不具合の想定は 5.3.3 節で詳述する．

テスト対象となるシステムの入力モデルを表 5.6 に示す．これらのシステムは，先行研究 [50] で実験対象として使用されていたことから本実験でも使用した．また

組み合わせテスト作成ツールとして，実テストで広く用いられている PICT を使用した．PICT によって得られた各システムの t-way テストスイートを構成するテストケース数を表 5.7 に示す．

5.3.3 実験準備

まず，テスト対象のシステムごとに t-way テストスイート（t = 2, 3, 4）を作成する．次に，各システムごとに不具合誘発スキーマとトリガスキーマをランダムに選択することで，想定するスキーマチェーンを設定する．ここで，不具合誘発スキーマとトリガスキーマの次数はどちらも 4 以下であり，不具合誘発スキーマとトリガスキーマは同一でないよう選択する．続けて，設定したスキーマチェーンの存在を想定した組み合わせテストの結果を生成する．生成には，設定したスキーマチェーンによる不具合の検出を最小限の網羅度 t で行える t-way テストを使用する．このような t の値は以下の方法で得られる．

- t=2 の場合から順に，t-way テストの結果を確認する．テストケースが一つでも失敗した場合は，その t-way テストの結果を実験に用いる．一方，全てのテストケースが合格した場合は，続けて (t+1)-way テストの結果を確認する．4-way テストでも全てのテストケースが成功した場合，設定したスキーマチェーンは使用せず，もう一度ランダムにスキーマチェーンを設定する．

表 5.1 に示した例題システムを用いて，本節で述べる組み合わせテスト結果生成の例を示す．制約を考慮して不具合誘発スキーマおよび対応するトリガスキーマをランダムに選択した結果として，5.2.3 節の適用例と同様に，不具合誘発スキーマとして (-, -, 0, 0) が，トリガスキーマとして (-, 2, 1, -) が選択されたとすると，これらがシステムに存在すると想定した場合の 2-way テスト結果として表 5.4 に示したものと同じテスト結果が得られる．このテストスイートのうち t_5, t_7, t_8 がテストに失敗したので，生成されたテスト結果は入力値限局能力を評価する実験に使用できる．また別の場合として，不具合誘発スキーマとして (0, 0, 1, -) が，対応するトリガスキーマとして (-, 2, 1, -) が選択された場合，用意した 2-way テストスイートの中のどのテストケースも失敗しないため，順に 3-way テストスイートおよび 4-way テストスイートについて，失敗するテストケースが存在するか確認する必要がある．

実験1のため，対象システムごとに，単一のスキーマチェーンの存在を想定した1,000個の組み合わせテスト結果を重複なく収集する．また実験2のため，対象システムごとに，2種類または3種類のスキーマチェーンの存在を想定した1,000個の組み合わせテスト結果を重複なく収集する．

5.3.4 比較手法

単純なOFOT法を比較手法として使用した．ただし，ここでは失敗したテストケースの一部のパラメータ値を変更した追加テストケースを前もって必要なだけ作成し，それらを連続して実行するという方法を取る．これは，順序依存性を持つ不具合誘発スキーマの特定には，少なくとも2つ以上のテストケースの連続した実行が必要なためである．

5.3.5 評価指標

準備した組み合わせテスト結果に提案手法と比較手法を適用し，以下の評価指標を得た．

特定精度

全ての正しいスキーマチェーンを特定した入力値限局の成功率を表す．以下の式によって計算される．

$$\text{特定精度} = \frac{\text{入力値限局成功数}}{\text{入力値限局実施件数（= 1,000）}}$$

本実験では，入力値限局は，その出力結果がそれまでに実行されたテストケースに含まれる想定したスキーマチェーンと全て一致する場合に成功と判断される．これは想定された複数のスキーマチェーンのうちの一部が，PICTによって設計されたテストスイート，および，入力値限局の過程で追加実行されるテストケースに一度も含まれない場合，そのスキーマチェーンを特定対象と見なさず，実行されたテストケースに含まれていたスキーマチェーンのみを特定対象として評価すべきという思想に基づく．

表 5.8: 実験 1 における結果：特定精度の比較

システム名	F-CODE	OFOT 法
Tomcat	1.000	0.083
Hsqldb	1.000	0.082
Gcc	1.000	0.095
Jflex	1.000	0.064
Tcas	1.000	0.100
Ave.	1.000	0.085

追加テストケース数

入力値限局を達成するために必要となる追加のテストケースの実行回数を表す.

特定再現率

一度の入力値限局において，無関係であるにも関わらず誤ってスキーマチェーンと特定されたものの少なさを表す．0 から 1 の値を取り，1 に近いほど正しいスキーマチェーンのみを特定結果として検出できたことを表す．この評価尺度は実験 2 でのみ使用される．以下の式によって計算される．

$$\text{特定再現率} = \frac{\text{全スキーマチェーン数}}{\text{出力されたスキーマチェーン数}}$$

5.4 評価結果

5.4.1 実験 1 に関する実験結果

表 5.8 に，単一のスキーマチェーンが存在する場合の各システムの組み合わせテスト結果を対象とした提案手法および比較手法の特定精度を示す．表 5.9 に，同様の場合における追加実行されたテストケース数の平均値，中央値，最小値，および最大値を示す．ここから，以下の結果を観察できる．

- F-CODE は全ての場合で単一のスキーマチェーンを完全に特定することができた．一方，比較手法である単純な OFOT 法の特定精度は，どのシステムの

表 5.9: 実験 1 における結果：追加テストケース数の比較

システム名	F-CODE				OFOT 法			
	Ave.	Med.	Min.	Max.	Ave.	Med.	Min.	Max.
Tomcat	43.1	37	33	175	30.5	20	10	410
Hsqldb	49.7	43	39	161	37.5	24	12	384
Gcc	42.6	37	33	149	32.3	20	10	490
Jflex	53.1	46	33	188	40.2	26	13	507
Tcas	85.7	49	33	1,541	116.7	48	12	6,504
Ave.	54.8	42.4	-	-	51.4	27.6	-	-

テスト結果を対象とした場合でも 0.1 以下にとどまった．

- F-CODE は Tcas を除くすべての対象システムで，比較手法よりも多くの追加テストケースの実行を必要とした．F-CODE が必要とする追加テストケース数は，単純な OFOT 法と比較して全体の平均で 107 %，中央値の平均で 154% となった．ただし，最大値は比較手法が遥かに高くなった．

5.4.2 実験 2 に関する実験結果

順序依存性による非決定的なテスト結果に対する比較手法のパフォーマンスが非常に低いことが実験 1 で示されているため，実験 2 では F-CODE にのみ焦点を当てる．表 5.10 に，スキーマチェーンが 2 個および 3 個存在する場合の F-CODE の特定精度および平均特定再現率を示す．表 5.11 に，同様の場合における F-CODE で追加実行されたテストケース数を示す．ここから，以下の結果を観察できる．

- 複数のスキーマチェーンが存在する場合，F-CODE は完全な特定を達成できなかった．またスキーマチェーンの数が増えるごとに特定精度が低下した． ただし，3 つのスキーマチェーンが存在する場合でも，特定精度は極端に低くなることはなく，たとえば 77 %を下回ることはなかった．

表 5.10: 実験 2 における結果：スキーマチェーンが n 個存在する場合の F-CODE の特定精度および平均特定再現率

システム名	特定精度		平均特定再現率	
	$n=2$	$n=3$	$n=2$	$n=3$
Tomcat	0.928	0.776	0.983	0.957
Hsqldb	0.923	0.796	0.991	0.954
Gcc	0.908	0.770	0.982	0.957
Jflex	0.929	0.793	0.981	0.958
Tcas	0.942	0.862	0.992	0.956
Ave.	0.926	0.799	0.986	0.956

- どの対象システムでも，平均特定再現率は 95 %を超えた．したがって，F-CODE の特定精度が低い場合でも，誤ったスキーマチェーンを特定結果として出力することはほとんどない．

- スキーマチェーンの数が増えると，追加実行されるテストケース数が増えた．実行数はスキーマチェーンが 1 つの場合と比較して，スキーマチェーンが 2 つの場合では平均 130%，3 つの場合では平均 162%増加した．

5.5 議論

5.5.1 比較手法の特定失敗に関する原因分析

実験 1 の結果から，F-CODE は全ての場合でスキーマチェーンを特定できたが，比較手法である単純な OFOT 法ではほとんどの場合で特定に失敗した．本節では，比較手法が特定に失敗した理由を分析する．

OFOT 法のメカニズム，および，5.3.4 節で示したその運用を考慮すると，使用した比較手法は次の理由から特定に失敗したと考えられる．

- 連続して実行される追加テストケースのうち最初のテストケースは常にテスト

表 5.11: 実験 2 における結果：スキーマチェーンが n 個存在する場合の F-CODE の追加テストケース数

システム名	$n = 2$				$n = 3$			
	Ave.	Med.	Min.	Max.	Ave.	Med.	Min.	Max.
Tomcat	54.4	43	33	262	66.8	68	33	243
Hsqldb	65.4	51	39	184	78.1	80	39	507
Gcc	56.2	43	33	272	67.8	68	33	418
Jflex	68.4	50	42	264	82.7	86	42	440
Tcas	110.8	91	39	800	149.7	134	39	2,330
Ave.	71.1	57.4	-	-	89.0	87.2	-	-

に合格することになる．これは，仮に最初のテストケースに不具合誘発スキーマが含まれていた場合でも，そのテストケースが失敗するためにはそれ以前にトリガスキーマを含むテストケースが実行されていることが必要となるからである． しかし，純粋な OFOT 法ではトリガスキーマの存在を想定していないためそれが保証されず，最初のテストケースに不具合誘発スキーマが含まれていたとしてもテストに合格することになる．したがって，最初の追加テストケースが失敗したテストケースの n 番目（通常は 1 番目）のパラメータの値を変更していた場合，失敗したテストケースの n 番目のパラメータの値が自動的に不具合誘発スキーマの一部と見なされてしまい，誤検出の発生原因となる．

- 不具合誘発スキーマを含む追加テストケースが失敗するためには，それ以前に実行されるの追加テストケースの中に対応するトリガスキーマが偶然に含まれる必要があるが，多くの場合これは低い確率でしか起こらない．OFOT 法において不具合誘発スキーマを含むにも関わらず失敗しないテストケースが存在する場合，誤検出の発生原因となる．

以上を理由として，単純な OFOT 法は実行順序に依存するテスト結果の入力値限局に適さないと言える．加えて，運良く不具合誘発スキーマを特定できたとしても，対応するトリガスキーマを特定するメカニズムを持たないため，不具合の発生

を再現する正確な条件が得られず，結果が無意味となる可能性が高い．

5.5.2 追加テストケース数に関する考察

表 5.9 に示す実験 1 の結果から，F-CODE および単純な OFOT 法が必要とするテストケースの追加実行数を比較すると，だいたい同程度の平均値および中央値となっていた．ただし多くの場合，F-CODE は単純な OFOT 法よりも 10 から 25 程度多くの追加テストケースの実行を必要とした．これは，F-CODE が，OFOT 法では表 5.3 に，F-CODE では表 5.5 のフェイズ 2 に示したような不具合誘発スキーマの特定作業に加えて，表 5.5 のフェイズ 1 に示すトリガスキーマの特定，およびフェイズ 3 に示すトリガスキーマを含むテストケースの特定にもテストの追加実行を必要とすることが原因である．フェイズ 1 における追加実行数はテストスイート内での失敗したテストケースの位置に依存し，フェイズ 3 における追加実行数はテストケースのパラメータ数に依存する．

次に，対象システムによる結果の差について考察する．実験に使用したテスト対象システムはパラメータ数にほぼ差がないため，提案手法のフェイズ 2 およびフェイズ 3 で発生する追加テストケース数にもあまり差は出ず，フェイズ 1 で発生する追加テストケース数に大きく依存することになる．前述の通り，この数はテストスイート内での失敗したテストケースの位置に依存する．したがって提案手法では，追加実行数の最大値および平均値はテストスイートを構成するテストケース数に依存することが分かる．例えば，表 5.9 における Tcas に F-CODE を適用した際の追加実行数の最大値である 1,541 は，表 5.7 に示した Tcas の 4-way テストスイートを構成するテストケース数である 1,382 という数に近く，使用した 4-way テストスイートの最後尾付近のテストケースが初めて失敗したことに由来することが分かる．一方で，比較手法では提案手法におけるフェイズ 2 の処理しか行わないため，追加実行数は単純にシステムのパラメータ数に依存する．ただし，比較手法は特定精度が低いため，他のテスト失敗を説明できるスキーマチェーンを見つけることができず，その場合は特定作業を何度も繰り返すことになる．最大値で比較した場合に比較手法が上回ったのはこのことが原因だと思われる．加えて，Tcas ではテストスイートに含まれるテストケース数が突出して多いため，最大値も高くなり，それによって Tcas

表 5.12: 原因 1・原因 2 の説明で用いる入力値限局の対象となるテストスイートおよびその結果

テストケース	p_a	p_b	p_c	p_d	結果
t_1	0	0	0	0	合格
t_2	0	1	1	1	合格
t_3	0	2	1	0	失敗
t_4	1	0	0	1	合格
t_5	1	1	0	0	合格
t_6	1	2	1	1	合格
t_7	2	0	1	1	合格
t_8	2	1	0	0	合格
t_9	2	2	0	0	合格

では比較手法の追加実行数の平均値が提案手法を上回ったのだと考えられる．

さらに実験 2 の結果から，特定すべきスキーマチェーンの数が増えると，F-CODE が要する追加実行回数も増えることも分かった．この単純な理由は，複数のスキーマチェーンを特定するためにフェイズ 1 から 3 の処理を複数回繰り返す必要があるためである．

5.5.3 複数の要因に対する提案手法の特定失敗に関する原因分析

実験 2 の結果から，複数のスキーマチェーンが存在する場合，F-CODE はその全てを特定できない場合があった．特定に失敗した数十件のケースを目視で確認し，失敗の原因を以下の 3 種類に分類した．説明の簡略のため，具体例の説明には表 5.1 に示した例題システムおよび表 5.2 に示したその 2-way テストスイートに置き換えたものを用いる．また不具合誘発スキーマを意味する *FI*，トリガスキーマを意味する *TR*，およびそのスキーマチェーン (*FI*, *TR*) を記号として用いる．

表 5.13: 原因 1 の説明で用いるフェイズ 2 で追加実行されたテストスイートおよびその結果

テストケース	p_a	p_b	p_c	p_d	結果
t_1	0	0	0	0	-
t_{a1}	1	2	1	0	失敗
t_{a2}	0	0	1	0	失敗
t_{a3}	0	2	0	0	失敗
t_{a4}	0	2	1	1	失敗

複数の不具合誘発スキーマが同じテストケースに含まれる場合

同じテストケースに複数の不具合誘発スキーマが含まれ，そのテストケースのみが失敗し，またそれらに対応するトリガスキーマが両方ともフェイズ 1 で特定されたテストケースに含まれている場合，F-CODE による特定は失敗する．これはフェイズ 2 において，OFOT 法によるパラメータ値の変更によって片方の不具合誘発スキーマが取り除かれたとしても，別の不具合誘発スキーマによってテストが失敗することで OFOT 法による特定を失敗させるためである．

たとえば，(FI_1, TR_1) および (FI_2, TR_2) の 2 つのスキーマチェーンが存在し，TR_1 および TR_2 は同一の (0, -, -, -) であるとする．また FI_1 が (0, 2, -, -)，FI_2 が (-, -, 1, 0) であるとする．この場合のテスト結果を表 5.12 に示す．F-CODE を適用すると，まずフェイズ 1 が適切に機能し t_1 が先に実行されることで t_3 が失敗することを特定する．次に，フェイズ 2 が表 5.13 に示すように動作したとする．t_{a1} と t_{a2} は FI_1 を含まないが，FI_2 を含むため失敗する．同様に，t_{a3} と t_{a4} には FI_2 を含まないが，FI_1 を含むため失敗する．t_{a1} から t_{a4} がすべて失敗したため，不具合誘発スキーマは (-, -, -, -) である，すなわち無条件でテストが失敗すると判断されるが，これは誤った結果となる．

この原因に対して効果的な改善を行うことは難しく，一度に複数のパラメータを改変して追加テストケースを設計するような，OFOT 法のメカニズムに代わる方法が必要である．

表 5.14: 原因 2 の説明で用いるフェイズ 2 で追加実行されたテストスイートおよびその結果

テストケース	p_a	p_b	p_c	p_d	結果
t_1	0	0	0	0	-
t_{a1}	1	2	1	0	合格
t_{a2}	0	0	1	0	失敗
t_{a3}	0	2	0	0	失敗
t_{a4}	0	2	1	1	失敗

OFOT 法の過程で新たな不具合誘発スキーマが混入する場合

フェイズ 2 において，t_{fail} の一部のパラメータ値を変更することで別の不具合誘発スキーマが含まれてしまい，かつその別の不具合誘発スキーマに対応するトリガスキーマがフェイズ 1 で特定された t_{trg} に含まれている場合，テスト結果は成功に変化せず，変更したパラメータ値が不具合誘発スキーマとは無関係であると判断されてしまうため，F-CODE は特定に失敗する．

前節の例と同様に，(FI_1, TR_1) および (FI_2, TR_2) の 2 つのスキーマチェーンが存在し，TR_1 および TR_2 は同一の (0, -, -, -) であるとする．一方で，FI_1 が (0, 2, -, -)，FI_1 が (-, 0, 1, 0) である場合を考える．この場合でも，テスト結果は前節の例と同様の表 5.12 に示す通りとなる．またこれも同様に，フェイズ 1 は適切に機能し t_1 が先に実行されることで t_3 が失敗することを特定できる．ここで，続くフェイズ 2 が表 5.14 に示すように動作した場合を考える．t_{a1} で p_a の値を 0 から 1 に変更すると FI_1 が削除されるため，テスト結果が合格に変わる．しかし，t_{a2} で p_b の値を 2 から 0 に変更すると，FI_1 は削除されるが，新たに FI_2 が t_{a2} に含まれるようになるため，結果は失敗のままとなる．また t_{a3} と t_{a4} は FI_1 を削除しないため失敗のままである．こうして t_{a2}，t_{a3}，t_{a4} が失敗したという結果から (0, -, -, -) が不具合誘発スキーマと判断されるが，これは誤った結果となる．

こちらも OFOT 法のメカニズムに由来する失敗原因であるため，こうした問題に効果的な別の入力値限局手法を開発し取り入れない限り，根本的な改善は不可能

表 5.15: 原因 3 の説明で用いる入力値限局の対象となるテストスイートおよびその結果

テストケース	p_a	p_b	p_c	p_d	結果
t_1	0	0	0	0	合格
t_2	0	1	1	1	失敗
t_3	0	2	1	0	合格
t_4	1	0	0	1	合格
t_5	1	1	0	0	失敗
t_6	1	2	1	1	合格
t_7	2	0	1	1	合格
t_8	2	1	0	0	失敗
t_9	2	2	0	0	合格

である．

発見済みのスキーマチェーンによって未発見のスキーマチェーンが隠される場合

F-CODE では効率を最大化するため，既に発見されたスキーマチェーンで説明できるテストの失敗に対する特定作業をスキップする．ただし，スキップされたテストケースに別の不具合誘発スキーマが含まれている場合，そちらについては出力されず，したがって完全な特定には至らない．

前節までの例と同様に，(FI_1, TR_1) および (FI_2, TR_2) の 2 つのスキーマチェーンが存在し，TR_1 および TR_2 は同一の (0, -, -, -) であるとする．また FI_1 が (-, 1, -, -) および FI_2 が (1, -, 0, 0) であるとする．この場合に得られるテスト結果を表 5.15 に示す．t_2，t_5，t_8 の 3 つのテストケースがテストに失敗している．このうち t_2 と t_8 には FI_1 のみが含まれ，t_5 には FI_1 と FI_2 の両方が含まれている．まず，t_2 を対象に特定作業を行う．その結果，t_2 を失敗に導いた (FI_1, TR_1) については問題なく特定される．しかしここで，t_5 および t_8 にも TR_1 が含まれているため，t_5 および t_8 のテスト失敗はどちらも (FI_1, TR_1) の存在によって説明できることになる．したがって F-CODE は t_5 および t_8 に対する特定作業をスキップし，(FI_1, TR_1) だけを報告して動作を完了す

る．しかし実際には t_5 には FI_2 も含まれており，またこちらは特定されていないため，ソフトウェアの修正によって (FI_1, TR_1) だけが誘発する不具合が取り除かれたとしても，t_5 は (FI_2, TR_2) が誘発する不具合によって失敗してしまう．

この原因による特定失敗はスキップ機能を無効にすることで改善できるが，その代償として追加テストケース実行回数は大幅に増加すると考えられる．

5.5.4 妥当性への脅威

実験に用いた提案手法 F-CODE および比較手法を実現するプログラムはいずれも著者が作成したものであり，実装に誤りがあった場合に結果に影響をもたらす可能性がある．したがって事前に試運転を行い，動作の正しさを入念に確認することで妥当性を高めた．

実験では，13 個を上限とする入力パラメータ数を持つシステムを対象とした 4-way テスト以下の t-way テストを用い，一度に最大 3 つのスキーマチェーンを想定したテスト結果を実験対象とした．そのため，これらの上限を上回る実験対象を用いた場合は異なる実験結果が得られる可能性がある．ただし，妥当性の確保のため，先行研究で用いられたものと同じ 5 種類のテスト対象システムを使用した．

また我々の提案および評価は，6.1.2 節に示した前提に基づいている．そのため，提案手法は前提に従わない順序依存性を持つ非決定的なテスト結果の入力値限局に適していない可能性がある．例えば他のバリエーションとして，一つの不具合誘発スキーマに対応するトリガスキーマが複数存在する場合や，トリガスキーマに依存せず単に実行回数に依存する場合が考えられる．ただし，我々の前提は実行順序への依存の形式として最も一般的なものであり，今回我々が提案したアイデアを適切に拡張することで個別に対応が可能であると考えている．

第 6 章

OSSを対象としたJavaテストコードの再利用可能性分析

6.1　テストコードの自動移植方法の構想

本節では，ある Java プロジェクトに存在するテストコードを別のプロジェクトに自動的に移植する方法，および，そうした移植が可能となる条件について構想する．

6.1.1　移植の方針

移植方法の方針を以下にまとめる．

- テストコードの移植はテスト機能の最小単位であるテストメソッドごとに行う．1 つのテストメソッドにつきそれを実行するための独立した 1 つのテストコードを作成する．

- はじめに，テストコードの移植元として使用するプロジェクトと移植先となるプロジェクトをそれぞれ解析する．移植可能なテストコードを発見した場合，移植先のプロジェクトで動作するよう調整を施したテストコードを作成し，移植する．

- テストコードの調整操作としてテストコード中の識別子の変更のみを許可する．これは調整操作を可能な限り単純化するためである．

- テストメソッドの実行可能性が移植先プロジェクトでも保たれるためには，テストメソッドが有するテストメソッド外部への依存関係を適切に解決する必要がある．テストメソッドが有する依存の対象は，次の (A) から (C) の 3 種類に分類できる．

  - (A) テスト対象である，プロダクトコードへの依存.
  - (B) テストメソッドと同じテストクラス内で定義されたプライベートメソッドなど，テストコード上で定義された箇所への依存.
  - (C) プロジェクトの外部からインポートされたサードパーティ製ライブラリへの依存.

  このうち，(B) と (C) については移植するテストメソッドとともに移植することで依存を保つことができる．したがって，(A) の依存を解決するようにテストコード中の識別子を変更することで，移植可能なテストコードが作成できる．逆に言えば，識別子の変更操作のみによって (A) の依存を解決することが可能なテストメソッドのみを本手法による移植の対象とする．テストメソッドが移植可能となる条件は第 6.1.3 節で詳述する．

- 移植が可能となる識別子の置換候補のうち，意味的に類似するものを選択することでテスト操作の意味的な適切さを担保する．これは「それぞれのテストコード内に存在するテスト対象を指す識別子名が意味的に類似していれば，より意味的に近いテストが行われている」というヒューリスティックに基づいている．

6.1.2 前提

テストメソッドの移植は，以下を前提とする．

- 移植元プロジェクトはプロダクトコードおよびテストコードを有している．移植先プロジェクトはプロダクトコードのみを有し，テストコードを有さない．
- 移植先プロジェクトおよび移植元プロジェクトはそれぞれの開発者が完成済みであると認識している．また，プロジェクトのコンパイルが可能な状態である．これは開発者の意図しない不具合がプロジェクトに存在することを許す．
- 再利用の対象となるテストコードは，移植元プロジェクトにおいて意図通り正しく動作する．
- 移植元プロジェクトにおいてテストコードの実行のために要求されているサードパーティ製ライブラリは，移植先プロジェクトでも同じように利用することができる．すなわちライブラリの非公開状態や，移植先プロジェクトが元々使用しているライブラリとの衝突は考えない．

6.1.3 移植可能条件

移植を試みるテストメソッドが移植先で実行可能性を保つことができる場合，移植が可能であると考える．テストメソッドが実行可能性を保つためには，プロダクトコードへの依存を保つ適切な識別子の置換が行える必要がある．したがって，置換可能な識別子を提供するプロダクトコード中の参照箇所が移植先プロジェクトに全て存在していることが移植可能条件となる．

テストコード側から見ると，プロダクトコードの各参照箇所はブラックボックスとして振る舞う．したがって，テストコードが依存するプロダクトコードで定義された箇所に対する入出力関係が表 6.1 に示すように移植元と移植先で保たれてさえいれば，動作上の問題はない．

ここでデータ型の一致について，プリミティブ型や，標準ライブラリおよびサードパーティ製のライブラリのクラスを参照する参照型は，完全一致する場合に限り一致とみなす．一方，移植元プロジェクトで定義された固有クラスへの参照型は，移

表 6.1: 入出力関係が保たれる条件

依存形式	条件
変数の参照	変数のデータ型が一致する
メソッドの呼び出し	メソッドにおける引数のデータ型とその並び, および返り値のデータ型が一致する
インスタンス生成	コンストラクタメソッドにおける引数のデータ型とその並び, および作成されるインスタンスのデータ型が一致する

表 6.2: 移植用テストコードにおけるプロダクトコードへの呼び出し例

クラス名	メソッド名	引数のデータ型	返り値のデータ型
ClassA	convert	[long]	int
ClassA	setBtoA	[ClassB]	void
ClassB	getValue	[ ]	int

植先プロジェクトで定義される任意の固有クラスへの参照型に一致するとみなす．ただし，こうした固有クラス間の依存関係が移植元と移植先で保たれる必要がある．

固有クラス間の依存関係について，具体例を用いて詳述する．あるテストメソッドを中心として関連参照箇所を抽出し作成した移植用テストコードを考える．表 6.2 は，このテストコードに存在する移植元プロジェクトのプロダクトコードへの依存を表している．すなわち，2 種類のクラスに属する 3 種類のメソッド呼び出しを通じて依存している．また下線は，移植用テストコード中に存在する置換すべき識別子であることを表している．ここで，ClassB は引数を持たず int 型を返すメソッド getB を有している．したがって，移植先にも同様の入出力を持つメソッドを有するクラスが存在する必要がある．加えて，ClassB は ClassA の持つメソッド setBtoA の引数に含まれる参照型に参照されている．したがって実際に移植先プロジェクトに要求されるのは，こうした ClassA と ClassB 間の関係と同じ関係を持つ 2 クラスの組み合わせとなる．

6.1.4 利用者とのインタラクション

本技術は利用者が開発するプロジェクトの情報をもとに，適用可能な全てのテストメソッドを既存プロジェクト群から検索し自動的にテストコードを生成することを目的とする．そのため必ずしも利用者が事前にテスト仕様を決定する必要はない．ただし，あらかじめテスト仕様をイメージしておくことで生成されたうち不必要なテストコードを除外することができる．また，場合によっては膨大な数のテストコードが生成されることも考えられる．この場合，利用者は候補として提示されたうち適切なテストコードを選択して使用することになる．将来的には生成数の上限を定めたり，よりテストしたい箇所をあらかじめ選択して重み付けすることも考えたい．また，ただ実行可能であるだけで意味的には適切でないテストが生成される場合もある．現状の提案内容では利用者の目視判断によって選択する必要があるが，将来的には 6.3.2 節で述べるような改善が考えられる．

6.2 OSS に存在するテストコードの調査

前節で述べた移植手法の開発に向け，その実現可能性を評価するため，以下に述べる 3 つの調査を行った．

6.2.1 RQ5：テストコードの移植元として使用できるプロジェクトはどの程度存在するか？

動機

テストコードからの移植を行う素材として，テストコードを含む既存のプロジェクトが多く必要となる．そうした Java プロジェクトがどれほど存在し利用できるかを把握することで，テストコードを再利用する手法がどれほど現実的かを調べる．またテストコードの移植において，共通のドメインに属するプロジェクトからの移植が有効であると考えられるため，得られたリポジトリのドメイン分類を行う．

調査対象

GitHub [3] に存在する全てのリポジトリを対象として，リポジトリ情報を保管するサービスである GHTorrent [29]，および GitHub が提供する GitHub API (v3) [4] を用いて絞り込む．また本実験で扱うテストコードは，テストフレームワークとして JUnit4, JUnit5, TestNG を使用したものに限定する．これらのフレームワークでは，メソッドに@Test アノテーションを付することがテストメソッドの基本的な識別ルールとなる．細かいルールも存在するが，本研究では簡単のため主たるものに統一した．GHTorrent のダンプデータは実験開始時に最新であった 2019 年 6 月に登録されたものを使用した．API による情報取得は 2019 年 9 月 2 日，リポジトリのダウンロードは 2019 年 11 月 15 日に実施した．

手法

調査 1-A から**調査 1-C** を順に行う．

調査 1-A

GHTorrent を用いて，GitHub に登録されている全てのパブリックなリポジトリのうち，以下全てを満たすものを抽出する．

1. 主な言語が Java である
2. 削除済みでない
3. フォークによって作られたものでない
4. スターを 100 以上持つ

3 番目および 4 番目の条件は低品質なプロジェクトを除外する目的で設定した．次に GitHub API を用いて，これらのリポジトリの master ブランチにおけるリポジトリ内のディレクトリ構造が記されたファイルである Tree オブジェクトファイルを参照することで,「test ディレクトリ下に存在する Java ソースファイル」の数を取得する．これらのファイルはテストコードであることが予想される．最後に，こうしたファイル数が 20 以上であるリポジトリをダウンロードし，ブランチが master でないものは master に切り替える．

調査 1-B

得られたリポジトリ内のファイルを静的に構文解析することで実際のテストコードを特定し，以下の条件のいずれかを満たすリポジトリを除外する．

- プロダクトコードが存在しない．ここでプロダクトコードとは，テストコードではない*.java ファイル全てを指す．
- テストメソッドが存在しない．ここでテストメソッドとは，テストコード中に存在する@Test アノテーションが付与されたメソッドを指す．また，@Ignore アノテーションの有無は関与しない．
- org.junit または org.testng を名称に含むライブラリを一度もインポートしていない．
- 構文解析を行う上で，デコードおよびパースが失敗し解析できなかった Java ファイルの割合が全体の 5%を上回る．あるいはこうした Java ファイルが 20 以上存在する．
- テストメソッドを含む Java ソースファイル数が 10 以下である．

上記の条件のうち最後の条件は，テストコードが極端に少ないリポジトリは移植元として明らかに不適なため，以降の調査から除外することを目的に設定した．静的解析には JavaParser [5] の Python ラッパーである javalang [9] を使用した．

調査 1-C

得られたリポジトリのそれぞれに対して著者がドメインのラベル付けを手動で行う．ラベルには，GitHub 上の優れた Java プロジェクトを紹介する Awesome Java [1] で用いられている 66 種類のドメインを参考に，著者の判断で 21 種類のドメインを使用した．各リポジトリの GitHub ページの description 文および README.md を目視で確認することで，それが属すると思われるドメインラベルを 3 つまで選択する．ドメインに繋がる情報が得られないものや判断がつかないものは No Category とする．

表 6.3: 調査 1-B で得られたリポジトリ情報

（単位：個）	最小値	平均値	中央値	最大値
スター数	100	1,121.6	329	81,817
全 Java ファイル	22	898.3	338	57,851
テストコード	10	216.3	63	4,508
プロダクトコード	1	676.7	233	57,372

結果

調査 1-A

全工程完了時，2,124 件のリポジトリが得られた．

調査 1-B

全工程完了時，1,862 件のリポジトリが得られた．これらのリポジトリの情報を表 6.3 に示す．リポジトリに含まれるテストコードの合計は 402,751 ファイルとなった．

調査 1-C

ドメイン分類の結果を表 6.4 に示す．各ドメイン毎にそのラベルが貼られたリポジトリ数を示しているため，数字の合計が総リポジトリ数と異なっている．ソフトウェア開発に用いられることを示す Development ドメインに属するリポジトリが最も多く，次いで，データの処理や規格を扱うことを示す Data ドメインや，Web アプリケーションであることを示す Web ドメインが多かった．したがって，これらのドメインに属するソフトウェアには，既存 OSS のテストコードを再利用しやすいと期待できる．

表 6.4: ドメイン分類結果

ドメイン	リポジトリ数	ドメイン	リポジトリ数
Development	339	Multimedia	100
Data	304	Testing	92
Web	256	Security	71
Network	201	Document	65
Distribution	159	Performance	46
Database	157	OS	44
Programing	137	Financial	41
Operation	135	Geospatial	31
Android	119	Middleware	30
Utility	104	Game	29
Science	101	No Category	54

6.2.2 RQ6：テストメソッドは他の参照箇所にどれほど依存しているか？

動機

テストメソッドが他の参照箇所，特にプロダクトコードで定義された参照箇所にどれほど依存しているかによって，その移植に要する条件は厳しくなる．こうした依存の大きさの傾向によって，移植可能であるために要求される条件の全体的な傾向を示す．

調査対象

RQ5 の調査で得られた 1,862 件のリポジトリのうち，上位 6 つのドメインに属するリポジトリをそれぞれスター数の多い順に 100 個選択し調査対象とする．これは，調査の規模と労力を抑えるため，再利用される可能性が高いリポジトリを代表させる意図による．このとき選択された中でファイル数が突出して多い 3 件のリポジトリは調査結果の妥当性を損ねると判断しこれらを除外した上で再選択した．ドメイン

表 6.5: 各テストメソッドが依存する参照箇所の集計および定義場所の分類結果

参照箇所	平均値 (個)	平均値の割合 (%)	中央値 (個)
全て	13.9	100.0	9
テストコード	1.8	12.9	1
プロダクトコード	4.0	28.8	2
外部ライブラリ	8.1	58.3	5

表 6.6: RQ3 での調査に用いる調査対象

ドメイン	移植先リポジトリ	テストメソッド数	移植元の合計テストメソッド数
Development	FreeTymeKiyan/LeetCode-Sol-Res	202	95,919
Data	plutext/docx4j	321	99,869
Web	apache/nutch	242	62,258
Network	mitreid-connect/OpenID-Connect-Java-Spring-Server	251	63,519
Distribution	alibaba/Sentinel	319	70,435
Database	xetorthio/jedis	213	60,481

間にリポジトリの重複があるため，選択されたリポジトリは合計で 509 個となった．

これらのリポジトリに含まれるテストコードからテストメソッドを抽出する際，extends または implements によって作成されたクラスおよび抽象クラスに含まれるテストメソッドを除外した．その結果，合計 348,045 個のテストメソッドが調査対象として得られた．

手法

静的構文解析によって，調査対象のテストメソッドが依存するテストコード内の参照箇所を全て特定し集計する．その後，それらの持つプロダクトコードあるいはサードパーティ製ライブラリへの依存を全て特定し集計する．これにより，調査対象のテストメソッドが依存する依存先種類の内訳を求める．

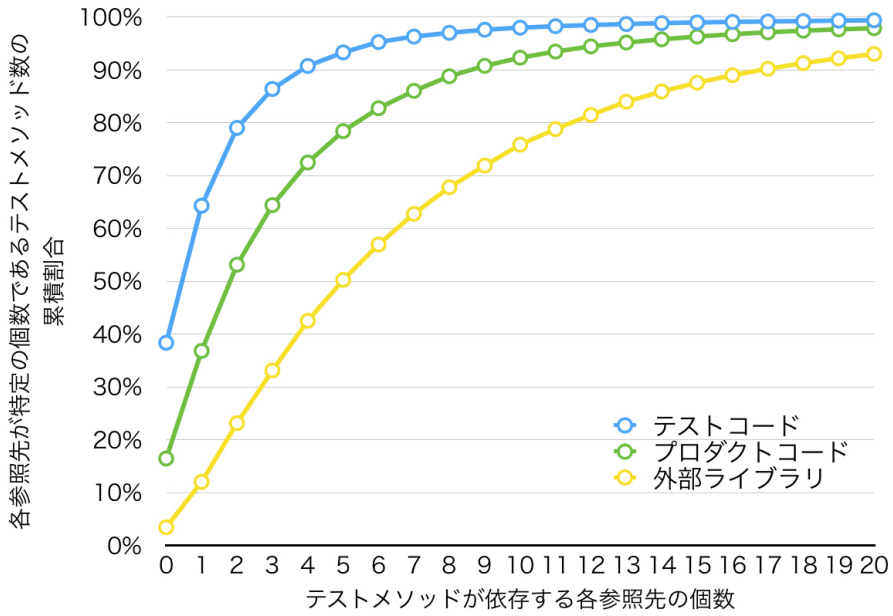

図 6.1: 各依存先種類への依存数が特定の個数であるテストメソッド数の累積割合

結果

テストメソッドの依存先を集計し分類した結果を表 6.5 に示す． 平均として，一つのテストメソッドは合計 13.9 個の参照箇所への依存を有している． そのうち，テストコード中の参照箇所に対しては平均 1.8 個の依存を有している．これは，これは，あるテストメソッドを移植する際に，自身の他に平均 1.8 個の参照箇所を同時に移植させる必要があることを示している．また，プロダクトコード中の定義ファクタに対して平均 4.0 個の依存を有している．これは，テストメソッドの移植を行う際に移植元プロジェクトと移植先プロジェクトの間で平均 4.0 個のプロジェクト固有の識別子の対応づけを行う必要があることを示している．

より詳細な分析のため，それぞれの依存先への依存が各個数をとるテストメソッドの個数の累積割合グラフを図 6.1 に示す． 例えば，緑のグラフで表されるプロダクトコード中の参照箇所への依存の個数が 1 のとき，テストメソッド数の累積割合は約 47%であることが読み取れる．これは，調査対象としたテストメソッドのうち約 47%が，プロダクトコード中の参照箇所への依存を高々1 個しか持たないことを意味している．

この図から次の結果が観察できる．

テストコードへの依存について．約 40%のテストメソッドは他のテストコード中の参照箇所への依存を持たず，それ単体で機能することがわかる． 他のテストコード中の参照箇所への依存が 4 つ以下のテストメソッドが全体の約 90%を占めており，大部分のテストメソッドは少量の補助コードを伴って移植用のテストコードを構成できる．

プロダクトコードへの依存について． プロダクトコード中の参照箇所への依存が 2 個以下であるテストメソッドが全体の約半数を占める．特に，約 20%のテストメソッドはプロダクトコードに対して単一の依存を持つ．これは EvoSuite 等のツールで作成可能な単純な単体テストに相当する．また，約 17%のテストメソッドはプロダクトコードへの依存を持たない．これらのテストメソッドは標準ライブラリやサードパーティ製ライブラリで実現できる機能のテストのみを行なっていると考えられる．

最後に，テストメソッドはプロジェクト固有ではないライブラリに対して平均 8.1 種類の呼び出しによる依存を有する． これらは，例えば JUnit の提供するアサーション用メソッドや，String クラス等の標準ライブラリが持つメソッドが多く該当した. 依存量の平均的な割合は最も高い. 移植先でもこれらのライブラリを使用できるよう設定できる前提のもとでは移植可能性に影響しないが，こうした依存量は設定作業にかかる労力の目安となる．

6.2.3 RQ7：既存テストコードには，テスト対象への依存関係が生成対象のテストメソッドと共通するテストメソッドはどれほど存在するか？

動機

移植元となるプロジェクトにおけるテストメソッドが持つプロダクトコードへの依存と，移植先のプロジェクトが本来必要とするテストメソッドが持つプロダクトコードへの依存が同一であるならば，テストメソッドを移植することで移植先で本来必要となるテストコードを生成できる可能性がある．特定のリポジトリを移植先およ

び移植元とした場合に，そうしたテストメソッドがどれだけ存在するかを調べることで，移植によるテストコードが達成できる効果の上限を示す．

調査対象

RQ6 への調査で使用した 6 ドメインの各 100 件のリポジトリのうち，テストメソッド数が中央値を取るものを移植先リポジトリとして，また他の 99 件を移植元リポジトリとして用いる．同ドメイン内で移植を行うことは移植条件において必須ではないが，少ないリポジトリから最大の効果が得られることが期待できる．各ドメインにおける移植先リポジトリのテストメソッド数，および移植元リポジトリの合計テストメソッド数を表 6.6 に示す．

手法

テストメソッドの移植先として選ばれたリポジトリが保有しているテストメソッドを，そのプロジェクトが必要とするテストメソッドの正解データ（以下，正解テストメソッド）として用いる．構文解析によって，それらのテストメソッドが持つプロダクトコードへの依存を抽出する．次に，移植元として扱うリポジトリ群の各テストメソッドについても同様に依存を抽出し，正解テストメソッドのいずれかと依存が一致するものを集計する．この依存の一致判定は 6.1.3 節で示したように，依存する参照箇所の入出力の一致，およびそれらのクラス間が持つ関係に基づく．

結果

結果を表 6.7 に示す．これは各ドメインの移植先リポジトリに存在する，プロダクトコードへの依存を 1 つ以上持つテストメソッド数 ($\#T_a$)，および，そのうち依存関係が一致するテストメソッドが移植元リポジトリに 1 つ以上存在するものの個数 ($\#T_b$)，またその割合 ($\#T_b/\#T_a$) を示している．移植先における正解テストメソッドのうち，平均して 83%，最も高いものでは 95%の割合で，プロダクトコードへの同一の依存関係を持つものが，移植元とした 99 件のリポジトリに 1 つ以上存在している．したがって，プロダクトコードへの依存関係が同一であるこれらのテストメソッドは，適切に識別子を改変することで移植先プロジェクトでも実行可能であることから，移

表 6.7: 移植するプロジェクト間でプロダクトコードへの依存関係が一致するテストメソッドの割合

ドメイン	$\#T_a$	$\#T_b$	割合
Development	147	140	0.95
Data	317	285	0.90
Web	236	206	0.87
Network	240	154	0.64
Distribution	304	209	0.69
Database	208	191	0.92
平均	-	-	0.83

植先リポジトリが本来必要とするテストメソッドを作成することができると考えられる．一方，残りの平均 17%では，プロダクトコードへの依存関係が一致するテストメソッドが移植元に存在しないため，移植による生成がそもそも不可能である．

6.3 議論

6.3.1 RQ への回答から得られる再利用によるテストコード自動生成の実現可能性への考察

RQ5 の結果から，1,862 件と決して少なくない数の Java OSS リポジトリがテストコードの再利用素材として利用できる．また今回の調査では，スター数が 100 以上とやや厳しい基準を設けているが，より少ないスター数でも別のメトリクスを併用することで高品質なテストコードを含む他のリポジトリを選定できる余地が残されている．また，時間とともに再利用に適したリポジトリが増えていくことも考えられる．したがって，本調査で得られた数を下限として，実際にはより多数の再利用素材の確保が見込める．こうした事実は，再利用によるテストコード自動生成の実現可能性を高める一因となる．

RQ6 の結果から，約半数のテストメソッドはプロダクトコードに高々2 種類の依

存を有しており，これらを適切に保つような移植元と移植先の間の識別子の対応関係を探すことで，実行可能性を保ったまま移植が可能であると言える．この結果は，テストコードの再利用の間口が広く設けられていることを示している．一方で，呼び出しが 9 種類以上とプロダクトコードへの複雑な依存を持つテストメソッドも約 1 割程度存在している．

RQ7 の結果から，移植先プロジェクトに本来必要となるテストメソッドについて，プロダクトコードへの依存の形が同一であるようなテストメソッドが既存プロジェクト群に高い割合で存在している．ただし，平均 17%，最も高いもので 36%のテストメソッドは，用意した同ドメインに属する 99 件の既存 OSS からは再利用によって生成され得ないことが示された．こうしたテストメソッドを再利用によって生成するには，用意するプロジェクトを増やすか，あるいは識別子の変更のみに限定しない再利用方法を考える必要がある．

以上 3 つの RQ の結果から，再利用の対象となりやすい既存テストコードが多く存在しており，また目的のテストコードを得られる可能性も高い．したがって，既存テストコードの再利用による自動生成手法の構想は現実的かつ有益であると考えられる．

6.3.2 自動生成に向けて解決すべき課題

現段階での提案内容では，移植したテストメソッドの実行可能性のみを議論の対象としており，意味的に適切なテストが移植できるか否かを問題としていない．そのため，意味的に適切なテスト操作を提供できるよう，テストメソッド中の識別子の対応づけ方法を改善する必要がある． ここで，テストメソッドが依存する参照箇所について，識別子名の意味的な類似度や，構文木の形状等を利用した処理内容の類似度などを算出し，候補を順位づけることで，より意味的に類似した識別子を探し出し，書き換えの対象とすることができる． こうすることで，移植元と移植先の間で意味的な類似性を保ちつつ適切なテスト操作手順を提供できると期待できる． そうした中で，どのような方法が最も効果的かを今後明らかにする必要がある．

また，移植したテストコードが適切なテスト操作手順を提供できる場合であっても，結果を照合するための適切な期待値を備えていない場合も考えられる． テスト

期待値の適切さを評価することは難しいため，利用者によって複数の候補から適切なものを選択したり，期待値部分を容易に変更できるような仕組みを整えることが求められる． さらに将来的には，本手法によって適切なテスト操作を，また他の手法によって適切なテストオラクルを検索し，両者を合成させることで自動的に適切なテストコードを作成することも考えられる．

6.3.3 妥当性への脅威

内的妥当性

本論文で実施した調査実験の大部分を，著者の作成したプログラムによって行っている．そのため，意図しないプログラムの動作不備が存在し実験結果に影響を与えている可能性は否定できない．また，RQ5 におけるリポジトリのドメイン分類は著者の判断かつシングルチェックで行われたため，理想的なドメイン分類とは異なっている可能性がある．また RQ7 の結果について，テストコード実行に必要な外部ライブラリのインポートが必ず成功するという前提の元での結果であり，この前提が事実と異なる場合には合致するテストメソッドが示された結果よりも少なくなる可能性がある．

外的妥当性

RQ5 の調査はある時点での GitHub 上の全ての Java リポジトリを対象としたため，外的妥当性への懸念は少ない．一方で，RQ6 での調査は特定の基準で選択された 509 個のリポジトリのみを，また RQ7 での調査は移植元と移植先のパターンとして各ドメインで 1 組のみを対象としているため，この調査結果を全てのリポジトリに対して一般化することは難しい．

第 7 章

再利用に基づく JAVA テストコード自動生成手法の提案

7.1 ツール仕様

本節では，第 6.1 節で提案したテストメソッド自動生成手法の構想を実行可能なツールとして実装するための詳細な仕様について述べる．本節で述べる仕様はプロトタイプ実装のひとつである．したがって第 6.1 節の構想を実現するためのより優れた仕様を検討する余地は大きく残されている．

テストコードの再利用素材として使用するプロジェクトを移植元プロジェクトと呼称する．また，移植元プロジェクトからテストコードを受け取るテスト対象のプロジェクトを移植先プロジェクトと呼称する．

ツールは，移植元プロジェクトおよび移植先プロジェクトのパスを入力として受け取り，移植先プロジェクトで実行可能なテストコードファイルのセットを出力する．移植元プロジェクトおよび移植先プロジェクトは，ツール使用者が事前に入手

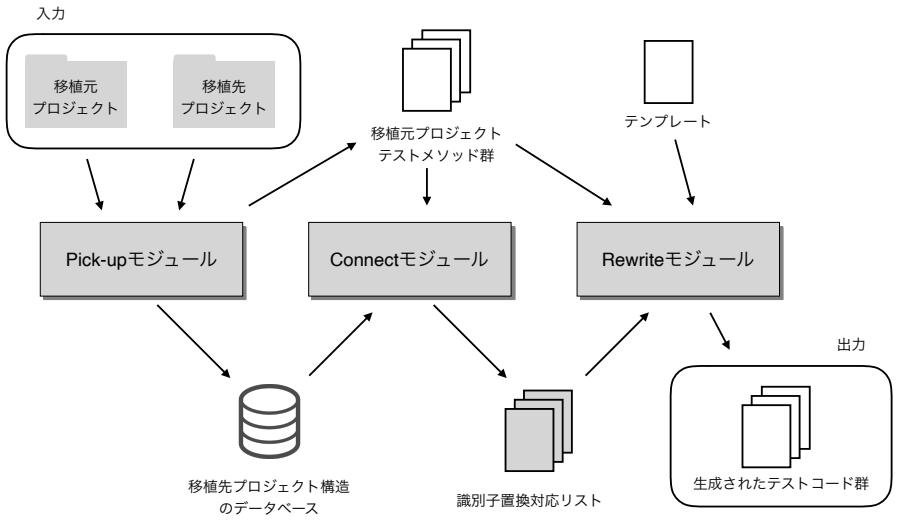

図 7.1: モデルにおける各モジュール間のデータフロー

しローカルに配置しておく必要がある．ツールの内部実装は，Pick-up モジュール，Connect モジュール，Rewrite モジュールに分けることができる．図 7.1 は，これらのモジュール間におけるデータフローの概要図を示す．それぞれのモジュールについて以下で説明する．

7.1.1 Pick-up モジュール

Pick-up モジュールの目的は，移植元プロジェクトおよび移植先プロジェクトの内容をそれぞれ解析し，移植に必要な情報を抽出することである．まず，移植元プロジェクトから全てのテストコードを抽出し，そこからテストメソッドを抽出する．次に，両方のプロジェクトのプロダクトコードを構成するクラスおよびそのメンバの情報を抽出し，データベース化する．最後に，移植元プロジェクトのテストメソッドが持つ依存を解析し，その定義場所を特定することで，テストメソッドが持つプロダクトコードへの依存を抽出する．

テストコード中の Public クラスで定義されているメソッドのうち，@Test アノテーションが付されたメソッドのみをテストメソッドとみなす．これは Java の主流なテストフレームワークである JUnit [7] および TestNG [13] における基本的な記法に基づいている．また依存解析が複雑になることから，他クラスを継承して作成さ

れたテストクラス内のテストメソッドおよび抽象クラス内のテストメソッドを無視する．加えて，@Ignore アノテーションが付されたテストメソッドも無視する．

ソースコードの依存解析には静的構文解析を使用する．静的構文解析には，代表的な Java の構文解析ツールである JavaParser [5] の Python ラッパーである javalang [9] を使用した．実行時の情報を用いた動的解析や，コンパイルして得られるクラスファイルのバイトコード解析は確実な方法であるものの，入手した多くの移植元プロジェクトをビルドし実行可能な状態にするのは困難な操作となる．一方，静的構文解析を用いることでプロジェクトをビルドすることなくソースコードを分析できるが，厳密に依存解析を行うことが難しい場合がある．

静的構文解析を用いて，テストメソッド中で行われるメソッド呼び出し，フィールド参照，インスタンス生成を全て抽出する．次に，それらが対象としているクラス名を特定する．その後，作成した移植元プロジェクトの構造データベースから「クラス名，要素名，引数の数」をクエリとして検索を行うことで，それらが定義されている場所を特定する．この操作によってその依存先がプロダクトコードか，テストコードか，サードパーティ製のライブラリなのかを分類する．加えてテストメソッドがテストコード内の他のメソッドに依存する場合，そのメソッドの持つ依存についても同様に依存先を特定する．最終的に得られた，テストメソッドの依存先となるプロダクトコード内の定義文を Connect モジュールに渡す．

7.1.2 Connect モジュール

Connect モジュールの目的は，テストメソッドが持つプロダクトコードへの依存のそれぞれを代替可能な移植先プロジェクトのプロダクトコードに対応づけ，置換が必要な識別子の置換先を特定することである．移植元プロジェクトのテストメソッドが持つプロダクトコードへの依存の中に，移植先プロジェクトに代替可能な依存先が存在しない依存がひとつでもある場合，そのテストメソッドは移植できないとみなす．またプロダクトコードへの依存を持たないテストメソッドも移植対象ではないとみなす．移植可能である場合，それぞれのテストメソッドについて，最も適切と思われる識別子の置換リストを一つだけ作成し，Rewrite モジュールに渡す．

代替可能なものを調べるために，テストメソッドにおけるプロダクトコードへの

依存をクラス毎にまとめ，全てのメンバに一致するメンバを含むクラスを移植先プロジェクトのプロダクトコードから検索する．メンバの一致は次のように判定する．メソッドの場合，引数の個数とそれぞれのデータ型が等しく，また返り値のデータ型が等しい．フィールドの場合，そのデータ型が正しい．データ型の一致は，プリミティブ型あるいはサードパーティライブラリから参照されたノンプリミティブ参照型の場合は完全な一致が要求されるが，プロジェクトの内部で定義された特有のクラスを参照するプリミティブ型の場合は，特有のクラス同士であれば一致するとみなす．ただし，クラス間の関係性を両方のプロジェクトで満たす必要がある．

さらに，識別子の置換候補を限定するために，自然言語処理技術を使用してより意味的に類似した識別子を選択する．本ツールでは，NLTK の WordNet を用いて，英語の語彙体系（Synonym）に基づく距離を計算することで意味的類似度を算出する．この方法は Word2Vec などの機械学習とベクトル化に基づく方法よりも高速に働くため採用している．複数語の単語からなる識別子はキャメルケースで分割し，最も語長の長い単語を選択することで代表させている．

7.1.3 Rewrite モジュール

Rewrite モジュールの目的は，移植元プロジェクトの移植可能なテストメソッドを基に，移植先プロジェクトで実行可能なテストコードを作成することである．新たなテストコードは図 7.2 に示すテンプレートを埋める形で作成される．テンプレートを埋める手順を以下に示す．

1. 移植対象のテストメソッドを (c) に挿入する．
2. そのテストメソッドが依存する同じテストクラス内のメソッド，変数，およびインナークラスを (b) および (d) に挿入する．
3. そのテストメソッドが依存する別のテストクラスを Private インナークラスとして (e) に挿入する．
4. Connect モジュールで得られた識別子の置換対応リストに従い，ここまで挿入したコードに含まれる識別子を置換する．

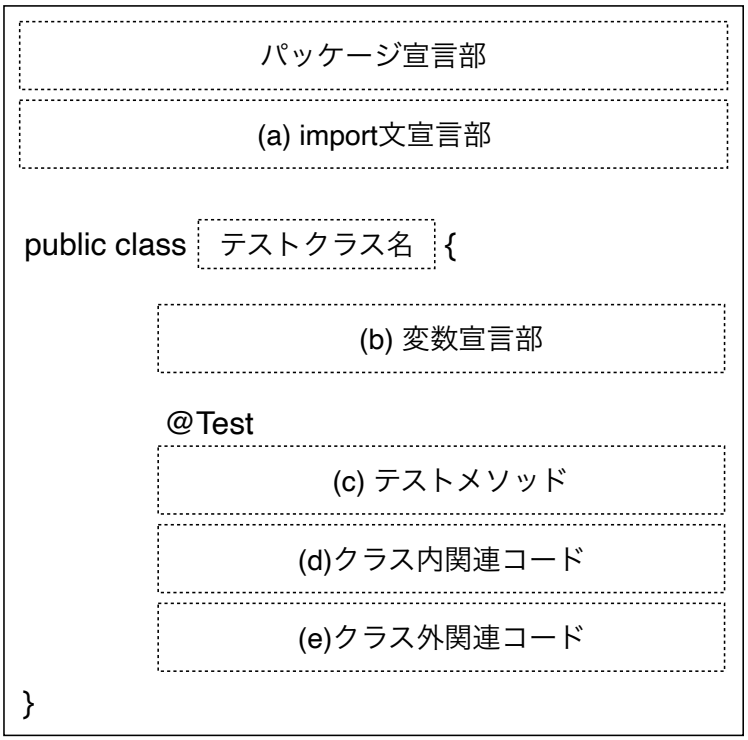

図 7.2: テストコードテンプレート

5. テストメソッドが依存するサードパーティ製ライブラリを対象とした import 文を (a) に挿入する．

6. 識別子を置換したことでテストメソッドの依存対象となった移植先プロジェクト内のプロダクトコードを対象とした import 文を (a) に挿入する．

終了後，テストコードのクラス名（ファイル名）を決定し，テストコードファイルを書き出す．

7.2 適用事例

本章では，試作したツールが実際に再利用可能なテストメソッドを検出し移植用テストコードを作成できることを実験的に示す．

7.2.1 適用手順

テストコードを生成したいプロジェクトを移植先プロジェクトとして用意する．テストコードを持つ数百程度のプロジェクトを移植元プロジェクトとして用意する．移植先プロジェクトと移植元プロジェクトの全てのペアに対して，7.1 章で説明した試作ツールを適用する．その結果，生成されたテストコードを調査することで，ツールの有効性を示す．

評価には GitHub に存在する，テストコードを有する Android アプリケーションのプロジェクトを使用する．Android アプリケーションのプロジェクトを使用する理由は，その多くが Java で実装されており，小規模かつ共通した要素を持つプロジェクト群であるためである．本手法では移植元プロジェクトとして扱うプロジェクトを限定していないため，利用可能な全てのプロジェクトを使用することが望ましい．しかし，実験のコストを妥当に抑えるため，使用するプロジェクトの数と規模を限定せざるを得なかった．より多いプロジェクト数を使用した場合の結果に近づけるため，移植先プロジェクトと移植元プロジェクトの間に共通した要素を持つことが望ましい．

生成されたテストコードから，生成されたテストコードの数，テストコードを移植した移植元プロジェクトの数，およびクラスカバレッジを取り出す．クラスカバレッジは，移植先プロジェクトのプロダクトコードに存在するクラスのうち，生成されたテストコードが依存するクラスの割合である．これらの情報は生成されたテストを実行することなく取り出すことが可能である．

7.2.2 データセット

Libraries.io [10] に 2020 年 1 月 12 日に登録されたデータベースを使用し，リポジトリを絞り込んだ．Libraries.io がモニタしている全てのリポジトリのうち，GitHub でホストされており，主なプログラミング言語が Java であるリポジトリは 27,727 個存在している．そのうち，Android アプリケーションに関連するものを選択するために，リポジトリ名・説明・キーワードに android (Android) と入ったものを選択する．同時に，最低限の品質保証のため，スター数が 1 以上のものを選択する．この結果，2,442 個のリポジトリが選択された．次にこのうち，test ディレクトリの下層

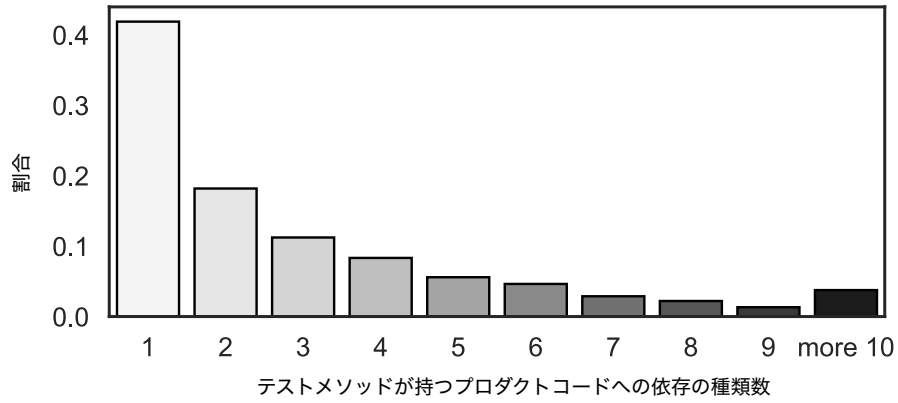

図 7.3: テストメソッドの持つプロダクトコードへの依存数の内訳

に存在する Java ファイルを 5 つ以上持つ 341 個のリポジトリを選択した．この操作は GitHub Web API を用いてディレクトリ構造が記された Tree オブジェクトファイルを参照することで達成された．最後に，リポジトリ内に存在する Java ソースコードファイルに対してソースコード解析を行い，テストメソッド数が 5 以下のリポジトリを除外した．また巨大なリポジトリを実験から除外するため，リポジトリのサイズが 10MB 以上のものを除外した．結果として，選択された 237 個のリポジトリを本実験に使用する．

これらのリポジトリにおけるリポジトリ内のテストメソッド数の中央値は 60 となった．したがって 60 を中心として，57 個から 62 個のテストメソッドを持つ 10 個のリポジトリを選択し，移植先プロジェクトとして使用する．また，残りの 227 個のリポジトリを移植元プロジェクトとして使用し，それぞれの移植先プロジェクトにテストコードを移植を試みる．表 7.1 は，移植先プロジェクトとして使用する 10 個のリポジトリの情報を示している．これらが持つプロダクトコード数の平均は 54.3，テストコード数の平均は 17.8 である．

また表 7.2 は移植元プロジェクトとして使用する 227 個のリポジトリの情報を示している．ここには合計で 27,505 個のテストメソッドが含まれる．このうち 21,742 個がプロダクトコードへの依存を 1 つ以上持つテストメソッドである．残りのテストメソッドはプロダクトコードへの依存が検出できなかったか，継承関係を含むなど Pick-up モジュールで除外されたテストメソッドである．テストメソッドの持つ

表 7.1: 移植先プロジェクトとして用いるリポジトリ

リポジトリ	プロダクトコード数	テストコード数	テストメソッド数
jeffdcamp/dbtools-android	133	16	57
StylingAndroid/Prism	57	28	58
facebook/TextLayoutBuilder	17	9	59
pwittchen/kirai	9	5	60
nitram509/jmacaroons	32	16	61
Microsoft/Telemetry-Client-for-Android	57	25	61
geri-m/soundtouch4j-api	49	18	61
citiususc/hipster	124	23	62
bertrandmartel/speed-test-lib	24	25	62
evant/loadie	41	13	62
平均	54.3	17.8	60.3

表 7.2: 移植元プロジェクトとして用いるリポジトリ群

リポジトリ数	227
全てのテストメソッド数	27,505
プロダクトコードへの依存を 1 つ以上持つテストメソッド数	21,742

プロダクトコードへの依存数の分布を図 7.3 に示す．この依存数の平均値は 2.99，中央値は 2 となり，これは第 6 章で得られた調査結果と比較してやや少ない．

7.2.3 結果

表 7.3 にそれぞれの移植先リポジトリに生成されたテストコード数，およびテストメソッドが 1 つでも移植に使用された移植元リポジトリ数を示す．各移植先プロジェクトについて，平均 953.6 個のテストコードが移植により生成された．移植先プロジェクトが本来有するテストコード数がおよそ 60 個であることを考えると，その約

表 7.3: 生成されたテストコード数と移植が行われたリポジトリ数

リポジトリ	生成されたテストコード数	移植が行われたリポジトリ数
jeffdcamp/dbtools-android	1400	144
StylingAndroid/Prism	807	101
facebook/TextLayoutBuilder	500	80
pwittchen/kirai	412	82
nitram509/jmacaroons	951	117
Microsoft/Telemetry-Client-for-Android	1421	135
geri-m/soundtouch4j-api	1115	108
citiususc/hipster	1047	113
bertrandmartel/speed-test-lib	881	105
evant/loadie	1002	105
平均	953.6	109.0

16 倍の数のテストコードが生成されたことが分かる．また 1 つ以上のテストメソッドを移植に提供した移植元リポジトリ数の平均は 109.0 個となった．この数のばらつきは少なく，どの移植先プロジェクトについても全体のおよそ半数の移植元リポジトリが移植元として機能していることが分かる．

表 7.4 に生成されたテストコードのクラスカバレッジを示す．表中の C_{all} は移植先プロジェクトのプロダクトコードで定義されている全てのクラスを指す．また C_{gen} は C_{all} のうち生成されたテストコードが依存する全てのクラスを指す．C_{org} は C_{all} のうち移植先プロジェクトが本来有しているテストコードが依存する全てのクラスを指し，$C_{gen\&org}$ は C_{gen} と C_{org} の共通部分を指す．生成されたテストコードが持つクラスカバレッジは平均で 47.8%となった．また，移植先プロジェクトが本来有しているテストコードが依存するクラスに限ったクラスカバレッジは平均で 39.4%となった．ここから，生成されたテストコードはテストすべきプロダクトコード内のクラスの約 4 割程度のみにしか，何らかの依存を持たないことが分かる．

表 7.5 に，各移植先プロジェクトにおける，移植されたテストメソッドが持って

表 7.4: 生成されたテストコードのクラスカバレッジ

リポジトリ	$\#C_{gen}/\#C_{all}$	$\#C_{gen\&org}/\#C_{org}$
jeffdcamp/dbtools-android	33.1%	39.1%
StylingAndroid/Prism	43.9%	14.6%
facebook/TextLayoutBuilder	47.1%	50.0%
pwittchen/kirai	44.4%	33.3%
nitram509/jmacaroons	40.6%	50.0%
Microsoft/Telemetry-Client-for-Android	64.9%	34.3%
geri-m/soundtouch4j-api	77.6%	75.0%
citiususc/hipster	31.5%	29.0%
bertrandmartel/speed-test-lib	45.8%	22.2%
evant/loadie	48.8%	23.1%
平均	47.8%	39.4%

いた置換可能なプロダクトコードへの依存数の内訳を示す．プロダクトコードへの依存が 1 種類のみであるテストメソッドの占める割合は，全ての移植先リポジトリについて 84.8%よりも多く，平均値は 93.8%となった．またプロダクトコードに 3 種類以上の依存を持つテストメソッドの割合は平均で 1.1%となった．ここから，移植に使用されているのは，プロダクトコードにごく簡単な依存を持つテストメソッドが大部分を占めることがわかる．

7.3 議論

7.3.1 ツールの能力評価

適用事例において生成されたテストコードに対して実行を介さずに得られたいくつかの評価指標から，本章で提案した試作ツールの能力を評価することを試みる．

本来必要なテストコードの約 16 倍の数のテストコードが生成された一方，実効的なクラスカバレッジは約 4 割に留まった．ここから，生成されたテストコードは

表 7.5: 移植されたテストメソッドが持つ置換可能なプロダクトコードへの依存数が n である割合

リポジトリ	$n = 1$	$n = 2$	$n \geq 3$
jeffdcamp/dbtools-android	84.8 %	10.8 %	4.4 %
StylingAndroid/Prism	95.2 %	4.6 %	0.2 %
facebook/TextLayoutBuilder	97.4 %	1.8 %	0.8 %
pwittchen/kirai	94.9 %	4.6 %	0.5 %
nitram509/jmacaroons	95.5 %	4.0 %	0.5 %
Microsoft/Telemetry-Client-for-Android	90.6 %	8.0 %	1.4 %
geri-m/soundtouch4j-api	96.0 %	3.8 %	0.3 %
citiususc/hipster	95.7 %	4.2 %	0.1 %
bertrandmartel/speed-test-lib	94.2 %	4.2 %	1.6 %
evant/loadie	93.9 %	5.3 %	0.8 %
平均	93.8%	5.1%	1.1%

そのうち本来必要でないテストコードが大部分を占めることが予想される．また，生成されたテストコードが依存するプロダクトコードのクラスが偏っていることが分かる．一方，プロダクトコードへの依存を一つ以上持つ 21,742 個のテストメソッドのうち平均して約 5%しか移植されないことから，どのようなテストメソッドからも無闇にテストコードが生成されているわけではない．移植に使用されるテストメソッドの傾向としては，プロダクトコードに 1 種類の依存しか持たないテストメソッドが約 94%を占めることが分かった．このことから，本ツールはテスト対象に対して簡単な依存しか持たないようなテストメソッドを主に再利用することが分かる．すなわち，結合テストやシナリオテストを実現するテストコードはほぼ生成されておらず，そのようなテストを本ツールのように簡単な操作で再利用することが難しいことを示唆している．

7.3.2 今後の課題

提案手法の評価における課題点として，次のことが挙げられる．生成されたテストコードは実行可能であることが求められる一方で，現時点ではテストコードを実際に実行したわけではないため，実際に実行可能なテストコードを生成できているのかが明らかではない．これは，移植先として使用するプロジェクトを手動で適切にビルドして実行可能な状態にするコストが高いこと，またサードパーティ製ライブラリへの依存の解決手段がまだ確立できていないことが原因である．今後，Docker や Ansible など半自動的に実行環境を整備できる技術を評価実験に組み込むことが考えられる．また，生成されたテストコードが実際にテスト対象のプロジェクトの内容に沿った意味のあるテストを生成できているのかについても明らかではない．これについても，テスト内容の目視確認以外による評価が難しく，生成された全てのテストの内容を確認するコストがが高いことが問題となっており，解決が求められる．

次にツール実装における課題点として，次のことが挙げられる．本ツールにおける依存解析には静的構文解析を使用したが，ルールベースによる依存参照先の特定は複雑になりやすく，解析結果の正確さが保証できない．しかし，動的解析やバイトコード解析を使用するためには無数に用意すべき移植元プロジェクトを全てビルドする必要があり，こちらもコストが高くなる．また，今回の適用事例では一つの移植先プロジェクトに対して平均 953.6 個のテストコードが生成されるなど，本来必要な量に対して大幅に多いテストコードが生成されるといった問題がある．ここには意味のない不適切なテストコードが多く含まれていると予想できるため，テストの適切さを何らかの方法でスコア付けするなど，本当に必要なテストコードを優先して生成するなどの工夫が必要である．また現在の仕様は各部分に最タスクに対して簡素な方法を採用したプロトタイプ実装であるため，より高度な方法でタスク解決を代替させる，あるいは新たなタスクで置き換えることが望まれる．例えば，クラスカバレッジの効率的な増加を目的として，次に移植元プロジェクトとして使用するリポジトリを逐次的に選択する仕組みなどが考えられる．

第 8 章

結論

8.1 達成事項

本論文では，ソフトウェアテストプロセスの横断的な自動化を妨げるいくつかの部分的な課題に対処するために，入力値限局およびテストコード生成の分野における新しい手法を提案した．また，実際のシステムに基づく評価実験あるいは大規模な事例調査によって各手法の有効性を示した．

まず，ロジスティック回帰分析によって得られる回帰係数を疑惑値として利用でき，既存手法よりも高精度に入力値限局が可能であることを評価実験により明らかにした．また限局すべき組み合わせの回帰係数が高くなることに加え，その部分集合にも高い回帰係数が見られることを発見した．この特性を利用することで，限局すべき組み合わせの部分集合を先に推定し，それらの超集合として組み合わせを推定する手法を構築した．また実際のシステムに基づくテスト結果を用いた評価実験によって，処理を高速化しつつ，高い精度で実際の不具合誘発入力値を推定できることを示した．

次に，先行研究で報告されたテスト結果が非決定的になる要因のうち実行順序へ

の依存性に焦点を絞り，非決定的なテスト結果に対応できるよう拡張された入力値限局手法を提案した．初めにテストの失敗を再現するテスト実行順序を特定し，それらを連続に実行して非決定性を排除することで，正しい限局結果を得ることが期待できる．実際のシステムに基づくテスト結果を用いた評価実験によって，提案手法の特定精度およびテストの追加実行回数の増加割合が好ましいものであることを示した．

最後に，OSS からの再利用による Java テストコードの自動生成手法の開発に取り組んだ．まず，特定のテストメソッドを実行可能性を保ちつつ自動的に移植するために必要な条件を定義し，それに基づいて OSS 中のテストメソッドが持つ依存関係等を調査することで，テストコードの再利用が現実的かつ有益であることを示した．加えて，既存テストコードの再利用によるテストコード自動生成モデルを実際に提案した．このモデルでは，テスト対象コードへの依存を分析し，機械的操作のみによって実行可能性を維持できるテストメソッドを検出することで，移植用のテストコードを自動的に生成する．さらに，自然言語処理技術によってテスト操作の意味的な適切さを担保する．また，このモデルをいくつかの Android アプリケーションに適用した事例によって，手法の有効性および課題を示した．

8.2 今後の課題

今後の課題として以下が挙げられる．

(1) 提案した入力値限局手法に関する課題

FROGa および FROGb では，その結果が好調であったことから，ロジスティック回帰の実装には単一のパッケージおよびそのデフォルトのパラメータ値のみを用いて評価に使用した．こうしたパッケージやパラメータの値を変更し最適化した場合に実験結果がどのように変化するか，また精度がより改善される場合があるかを詳細に調査することが望まれる．

F-CODE では，複数の不具合原因がある場合に誤った結果を返す場合があることが示された．複数の不具合原因の存在により対応できるように，また必要な追加テストケースの数を減らすために，アルゴリズムを改善することが望まれる．また，

実行順序への依存以外の形でテスト結果が非決定的となる他の要因についても同様に限局可能となる方法を考案し，複数の要因が混在している場合にも限局が可能となるようなフレームワークを模索していきたい．

またこれらの手法の評価には，不具合を誘発する入力値を想定して人工的に生成されたテスト結果を用いた．これは評価の妥当性の上では問題ないと考えているが，不具合箇所の修正に限局結果をフィードバック可能かどうかを調査するためにも，実際のシステムのソースコードに不具合を混入させたものをテストした結果を使用するなど，より実際の障害に近づけた評価を今後行いたい．

(2) 提案したテストコード再利用手法に関する課題

第 7.3.2 節で述べたように，生成されたテストコードが実際に実行可能か，また意味的に適切かを評価する仕組みを考える必要がある．また提案手法の改善案として，より精度の高い依存解析を実現すること，優先して作成すべきテストコードを評価すること，部分タスクのためにより高度な技術を導入することが挙げられる．

(3) テストプロセス全体の自動化に関する課題

本研究では入力値限局およびテストコードの自動生成の 2 つに焦点を当てて研究を行ったが，たとえこれらについて決定的に有効な手法が確立したとしても，テストプロセス全体の横断的な自動化には未だ不十分であると考えられる．依然としてボトルネックになるテスト中の部分プロセスを特定するために，既存のテスト自動化技術を用いたツールチェインによる自動化フレームワークを構築したい．

参考文献

[1] Awesome java. https://github.com/akullpp/awesome-java.

[2] Cucumber. https://cucumber.io.

[3] Github. https://github.com.

[4] Github api (v3). https://developer.github.com/v3/.

[5] Javaparser. http://javaparser.org.

[6] Jbehave. https://jbehave.org.

[7] Junit. https://junit.org.

[8] Junit5: A first test case. https://junit.org/junit5/docs/current/user-guide/#writing-tests.

[9] lavalang: Pure python java parser and tools. https://github.com/c2nes/javalang.

[10] Libraries.io.

[11] Pict: Pairwise independent combinatorial tool. http://github.com/Microsoft/pict.

[12] scikit-learn. http://scikit-learn.org/.

[13] Testng. https://testng.org.

[14] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proc. of 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-PT)*, pages 263–272, 2017.

[15] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. `https://arxiv.org/abs/1611.01989`, 2016.

[16] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object oriented metrics as quality indicators. *IEEE Trans. on Software Engineering*, 22(10):751–761, 1996.

[17] J. Bonn, K. Foegen, and H. Lichter. A framework for automated combinatorial test generation, execution, and fault characterization. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 224–233, 2019.

[18] L. C. Briand, W. L. Melo, and J. Wust. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. on Software Engineering*, 28(7):706–720, 2002.

[19] C. J. Colbourn and D. W. McClary. Locating and detecting arrays for interaction faults. *Journal of combinatorial optimization*, 15(1):17–48, 2008.

[20] C. J. Colbourn and V. R. Syrotiuk. On a combinatorial framework for fault characterization. *Mathematics in Computer Science*, 12(4):429–451, Dec 2018.

[21] J. Cramer. The Origins of Logistic Regression. Tinbergen Institute Discussion Papers 02-119/4, Tinbergen Institute, Dec. 2002.

[22] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios.

[23] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 177–188. ACM, 2009.

[24] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[25] A. Gargantini, J. Petke, and M. Radavelli. Combinatorial interaction testing for automated constraint repair. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 239–248, March 2017.

[26] L. S. Ghandehari, J. Chandrasekaran, Y. Lei, R. Kacker, and D. R. Kuhn. Ben: A combinatorial testing-based fault localization tool. In *Proc. of IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–4, 2015.

[27] L. S. G. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker. Identifying failure-inducing combinations in a combinatorial test set. In *Proc. of IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 370–379, 2012.

[28] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè. Automatic generation of oracles for exceptional behaviors. In *Proc. of 25th International Symposium on Software Testing and Analysis (ISSTA)*, pages 213–224, 2016.

[29] G. Gousios. The ghtorent dataset and tool suite. In *Proc. of 10th Working Conference on Mining Software Repositories (MSR)*, pages 233–236, 2013.

[30] J. D. Hagar, T. L. Wissink, D. R. Kuhn, and R. N. Kacker. Introducing combinatorial testing in a large organization. *Computer*, 48(4):64–72, Apr 2015.

[31] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25(5):45–52, 2008.

[32] R. Jayaram and R. Krishnan. Approaches to fault localization in combinatorial testing: A survey. In S. C. Satapathy, V. Bhateja, and S. Das, editors, *Smart Computing and Informatics*, pages 533–540, Singapore, 2018. Springer Singapore.

[33] H. Jin, T. Kitamura, E. Choi, and T. Tsuchiya. A satisfiability-based approach to generation of constrained locating arrays. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops*, pages 285–294, 2018.

[34] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

[35] C. J. Kapser and M. W. Godfrey. “cloning considered harmful”considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[36] S. K. Khalsa and Y. Labiche. An orchestrated survey of available algorithms and tools for combinatorial testing. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 323–334, 2014.

[37] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. pages 91– 95, 01 2003.

[38] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to combinatorial testing*. CRC Press, 2013.

[39] J. Li, C. Nie, and Y. Lei. Improved delta debugging based on combinatorial testing. In *2012 12th International Conference on Quality Software*, pages 102–105, Aug 2012.

[40] N. Li, A. Escalona, and T. Kamal. Skyfire: Model-based testing with cucumber. In *Proc. of 9th International Conference on Software Testing, Verification and Validation (ICST)*, pages 393–400, 2016.

[41] Y. Lu, S. Chaudhuri, C. Jermaine, and D. Melski. Program splicing. In *Proc. of 40th International Conference on Software Engineering (ICSE)*, pages 338–349, 2018.

[42] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, 2014.

[43] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[44] C. Martínez, L. Moura, D. Panario, and B. Stevens. Locating errors using elas, covering arrays, and adaptive testing algorithms. *SIAM Journal on Discrete Mathematics*, 23(4):1776–1799, 2009.

[45] J. Micco. Flaky Tests at Google and How We Mitigate Them.

[46] M. Motwani and Y. Brun. Automatically generating precise oracles from structured natural language specifications. In *Proc. of 41st International Conference on Software Engineering (ICSE)*, pages 188–199, 2019.

[47] T. Nagamoto, H. Kojima, H. Nakagawa, and T. Tsuchiya. Locating a faulty interaction in pair-wise testing. In *Proc. of IEEE 20th Pacific Rim International Symposium on Dependable Computing*, pages 155–156, 2014.

[48] C. Nie and H. Leung. The minimal failure-causing schema of combinatorial testing. *ACM Trans. Softw. Eng. Methodol.*, 20(4):15:1–15:38, Sept. 2011.

[49] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11, 2011.

[50] X. Niu, n. changhai, H. K. N. Leung, Y. Lei, X. Wang, J. Xu, and Y. Wang. An interleaving approach to combinatorial testing and failure-inducing interaction identification. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[51] X. Niu, C. Nie, Y. Lei, and A. T. S. Chan. Identifying failure-inducing combinations using tuple relationship. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 271–280, March 2013.

[52] A. Orso and G. Rothermel. Software Testing: A Research Travelogue (2000–2014). In *Proc. of Future of Software Engineering (FOSE)*, pages 117–132, 2014.

[53] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Proc. of 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA)*, pages 815–816, 2007.

[54] C.-Y. J. Peng, K. L. Lee, and G. M. Ingersoll. An introduction to logistic regression analysis and reporting. *The Journal of Educational Research*, 96(1):3–14, 2002.

[55] L. Sh. Ghandehari, Y. Lei, R. Kacker, D. R. R. Kuhn, D. Kung, and T. Xie. A combinatorial testing-based approach to fault localization. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[56] K. Shakya, T. Xie, N. Li, Y. Lei, R. Kacker, and R. Kuhn. Isolating failure-inducing combinations in combinatorial testing using test augmentation and classification. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 620–623, 2012.

[57] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser. How do automatically generated unit tests influence software maintenance? In *Proc. of 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 250–261, 2018.

[58] J. Shi, M. B. Dwyer, and M. B. Cohen. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34:633–650, 06 2008.

[59] L. Shi, C. Nie, and B. Xu. A software debugging method based on pairwise testing. In *Proc. of International Conference on Computational Science (ICCS), volume 3516 of Lecture Notes in Computer Science*, pages 55–81, 2005.

[60] D. Sondhi, D. Rani, and R. Purandare. Similarities across libraries: Making a case for leveraging test suites. In *Proc. of 12th International Conference on Software Testing, Validation and Verification (ICST)*, pages 79–89, 2019.

[61] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *Proc. of IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 260–269, 2012.

[62] Z. Wang, B. Xu, L. Chen, and L. Xu. Adaptive interaction fault location based on combinatorial testing. In *Proc. of IEEE 10th International Conference on Quality Software (QSIC)*, pages 495–502, 2010.

[63] C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 32(1):20–34, 2006.

[64] C. Yilmaz, E. Dumlu, M. B. Cohen, and A. Porter. Reducing masking effects in combinatorialinteraction testing: A feedback drivenadaptive approach. *IEEE Transactions on Software Engineering*, 40(1):43–66, Jan 2014.

[65] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[66] J. Zhang, F. Ma, and Z. Zhang. Faulty interaction identification via constraint solving and optimization. In *Proc. of International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 186–199, 2012.

[67] Z. Zhang and J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing. In *Proc. of ACM 20th International Symposium on Software Testing and Analysis (ISSTA)*, pages 331–341, 2011.

[68] W. Zheng, X. Wu, D. Hu, and Q. Zhu. Locating minimal fault interaction in combinatorial testing. *Advances in Software Engineering*, 2016:10, 2016.

[69] 情報処理推進機構. **ソフトウェア開発データ白書** *2018-2019*.